Office de la Propriété Intellectuelle du Canada
Un organisme d'Industrie Canada

Canadian Intellectual Property Office
An agency of Industry Canada

CA 2310186 A1 2001/12/02

(21) 2 310 186

(12) DEMANDE DE BREVET CANADIEN
CANADIAN PATENT APPLICATION

(13) A1

(22) Date de dépôt/Filing Date: 2000/06/02

(41) Mise à la disp. pub./Open to Public Insp.: 2001/12/02

(51) Cl.Int.^{7}/Int.Cl.^{7} H03M 13/31, H04L 1/22

(71) Demandeurs/Applicants:
KSCHISCHANG, FRANK, CA;
CASTURA, JEFFREY P., CA

(72) Inventeurs/Inventors:
KSCHISCHANG, FRANK, CA;
CASTURA, JEFFREY P., CA

(74) Agent: STRATTON, ROBERT P.

(54) Titre : METHODE ET SYSTEME DE DECODAGE
(54) Title: METHOD AND SYSTEM FOR DECODING

(57) Abrégé/Abstract:
Reduced complexity decoding algorithms for Low Density Parity Check codes are presented. The performance of these algorithms is optimized using the concept of density evolution and they are shown to perform well in practical decoding situations. The codes are examined from a performance vs. complexity point of view, and it is shown that there is an optimal complexity for practical decoders beyond which performance will suffer. The idea of practical decoding is used to develop the sum-transform-sum algorithm, which is very well suited for a fixed- point hardware implementation. The performance of this decoder approaches that of the sum-product decoder, but is much less complex.

ABSTRACT

Reduced complexity decoding algorithms for Low Density Parity Check codes are presented. The performance of these algorithms is optimized using the concept of density evolution and they are shown to perform well in practical decoding situations. The codes are examined from a performance vs. complexity point of view, and it is shown that there is an optimal complexity for practical decoders beyond which performance will suffer. The idea of practical decoding is used to develop the sum-transform-sum algorithm, which is very well suited for a fixed- point hardware implementation. The performance of this decoder approaches that of the sum-product decoder, but is much less complex.

Contents

List of Figures

List of Tables

List of Acronyms

APP	A Posteriori Probability
AWGN	Additive White Gaussian Noise
BCJR	Forward-Backward Algorithm (B.C.J.R. are the names of its inventor)
BER	Bit Error Rate
BSC	Binary Symmetric Channel
CDMA	Code Division Multiple Access
FBA	Forward-Backward Algorithm
FEC	Forward Error Correction
GA	Genetic Algorithm
LDPC	Low Density Parity Check
LLR	Log Likelihood Ratio
LLDP	Log Likelihood Difference Pair
MAP	Maximum A-Posteriori
SNR	Signal to Noise Ratio
SXS	Sum-Transform-Sum Algorithm

Chapter 1

Introduction

1.1 History

Communication systems of all types have to deal with the presence of noise over the communications channel, and it is the goal of system designers to mitigate the effect of noise in the system as efficiently as possible. To this end, many communications devices utilize some sort of forward error correction (FEC) code, which entails the addition of controlled redundancy to a transmitted signal in the hope that this structure will allow errors caused by the noisy channel to be detected and corrected.

Shannon put bounds on the signal-to-noise ratio (SNR) that is required to communicate reliably over an additive white Gaussian noise (AWGN) channel, and for a certain rate R of transmission, this Shannon limit is [22]

$$\frac{E_b}{N_0} \geq \frac{2^{2R}-1}{2R}, \tag{1.1}$$

where R is defined as the ratio of information bits to transmitted symbols. Traditionally, codes such as Hamming codes, BCH codes, and Reed-Solomon codes with algebraic decoding algorithms, and convolutional codes with trellis-based decoding algorithms have been employed to achieve gains in overall system performance. In recent years however, there has been a great deal of interest in iterative decoding schemes. Based on the idea of using very powerful and complex codes and utilizing sub-optimal decoding methods, they perform much better than any other class of code known. In fact, codes based on iterative decoding

can perform very well at SNRs that are within a fraction of a decibel of the Shannon limit.

There are two common types of iteratively decoded codes: low density parity check codes and turbo codes. Low density parity check codes (LDPC codes) were first introduced by Gallager in his PhD thesis in 1960 [6], and the performance of these codes was better than most other codes known at the time. Unfortunately, it was too difficult to implement the decoding algorithm for practical applications given the technology of the day. Indeed, Gallager was only able to simulate LDPC codes for short block lengths and relatively high probability of bit error on the IBM7090 computer he had available to him. Thus, due to the relative complexity, and the excitement at the time around convolutional codes, Gallager's LDPC codes remained relatively unmentioned for the next thirty years [28].

Berrou et al. [2], astounded the coding community in 1993 with their discovery of turbo codes, also known as parallel concatenated convolutional codes. The performance of these codes was shown to come to within less than 1dB of the Shannon limit when using *iterative* decoding. This remarkable improvement over the previously best known codes renewed interest in the search for other Shannon limit approaching codes and iterative decoding schemes.

MacKay was the first to show in 1996 that LDPC codes of long block length are very good and that their performance is close to, though not quite as good as, turbo codes [14]. Since then, a great deal of interest has been shown in these codes and a number of improvements and advancements have pushed their performance closer to the Shannon limit. Some of these advancements include LDPC codes over $GF(2^m)$ [4], irregular LDPC codes [12], and techniques to analyze the behaviour of different LDPC codes decoding algorithms [19, 18].

Recently, results have been presented showing that some constructions of LDPC codes actually outperform turbo-codes. For example, Spielman gave experimental results for a rate 1/2 irregular LDPC code that achieves a bit error probability of 3×10^{-5} for an E_b/N_0 of 0.7 dB [26]. Richardson and Urbanke have also reported similar results [18]. In fact, they have reported a construction for highly irregular LDPC codes of extremely long block length that achieves an asymptotically good performance to within 0.06dB of the Shannon limit for the AWGN channel. This is a remarkable result and an important milestone in the development of error control codes.

1.2 An Introduction to LDPC Codes

LDPC codes, despite the long name, are simply binary linear block codes based on a sparse parity check matrix. A code may be defined as an LDPC code if the average weight of rows and columns in the parity check matrix is small. As linear block codes, they have well defined generator and parity check matrices **G** and **H** respectively, and like the case for other binary linear block codes, the parity check matrix is used to decode a noise-corrupted codeword.

As Shannon's Channel Coding Theorem shows, good codes can be found by randomly choosing a codebook, and using a very large block length [22]. These very good codes can achieve an arbitrarily low probability of bit error for any rate below capacity. Unfortunately, Shannon did not give any constructive and practical methods for encoding and decoding these codes, and this has been the problem that coding theorists have since attempted to resolve [3].

Gallager developed LDPC codes with the benefits of randomness while maintaining a practical encoding and decoding scheme. His LDPC codes are created by randomly creating a large sparse parity check matrix **H** with uniform row and column weight, both of which are small to keep **H** sparse. By keeping the matrix sparse, the complexity of decoding is kept within reasonable limits.

In the realm of graph theory, this corresponds to a random bipartite graph that has a constant small number of edges terminating on both variable nodes and check nodes. This constraint of uniform variable and check node degree gives rise to what is known as *regular* LDPC codes, which is the original construction Gallager outlined.

It is a simple matter to convert a parity check matrix to a bipartite graph. Each row in **H** is represented by a parity check node, and each column is represented by a variable node, and these two types of nodes are placed across from each other on the graph. For every element in the binary matrix **H** that is a 1, an edge is drawn connecting the corresponding check node (row) and variable node (column). Let d_v be the degree of the variable nodes (weight of rows in **H**), d_c the degree of the check nodes (weight of columns in **H**), and n be the length of a codeword (number of columns in **H**). A regular LDPC code C then can be described as an $\{n, d_v, d_c\}$ LDPC code.

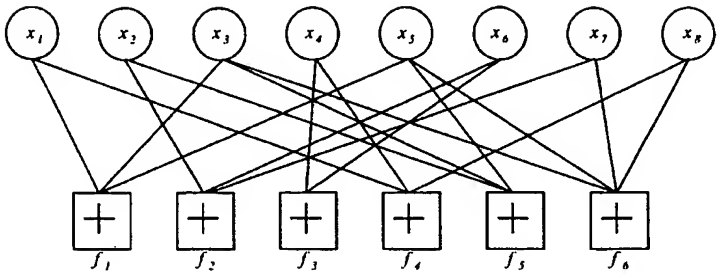

Figure 1.1: Factor graph representation of a LDPC code

It is also possible to create *irregular* LDPC codes. These codes differ from regular codes by the fact that the weight of the columns and rows in **H** are not equal for all columns or rows. For example, some columns may have weight 3, while others may have weight 4 or 5. The same holds true for the weights of the rows. Well chosen irregular codes have been shown to perform better than their regular counterparts [14]. The degrees that arise in the graph can be described by the *degree sequence polynomials* $\rho(x)$ and $\lambda(x)$ respectively, where

$$\lambda(x) = \sum_i \lambda_i x^{i-1} \text{ and } \rho(x) = \sum_i \rho_i x^{i-1}, \tag{1.2}$$

in which λ_i $(resp. \rho_i)$ represents the fraction of edges which are incident on a variable (resp. check) node of degree i. It follows that $\lambda(1) = \rho(1) = 1$. These polynomials sufficiently describe an *ensemble* of codes that may be generated from a given $\lambda(x)$ and $\rho(x)$; they do not describe a single construction. Note that throughout the remainder of this paper, the above polynomials are used in place of their integer counterparts d_v and d_c when discussing irregular LDPC codes.

Example: For the graph in Figure 1.1,

$$\lambda(x) = \frac{1}{3}(2x + x^2) \text{ and } \rho(x) = \frac{1}{9}(x + 6x^2 + 2x^3) \tag{1.3}$$

A lower bound on the rate R of an LDPC code can be determined from the degree sequence polynomials that describe it. Let e be the number of edges in the graph. The number of variable nodes of degree i is $\frac{e\lambda_i}{i}$. Therefore the total number of variable nodes n

is the summation over all degrees and is thus

$$n = \sum_i \frac{e\lambda_i}{i} = e\int_0^1 \lambda(x)dx. \tag{1.4}$$

Similarly the total number of check nodes is

$$r = \sum_i \frac{e\rho_i}{i} = e\int_0^1 \rho(x)dx. \tag{1.5}$$

Since the dimension of the code is at least $k \geq n - r$ with strict inequality if some of the r check nodes are linearly dependent, we find

$$R = \frac{k}{n} \geq 1 - \frac{r}{n} = 1 - \frac{\int_0^1 \rho(x)dx}{\int_0^1 \lambda(x)dx}. \tag{1.6}$$

1.3 Generalizations

There are many areas of interest to researchers studying LDPC codes. Since LDPC codes are linear block codes, it is possible to define them over a larger field, $GF(q)$ for $q > 2$. LDPC codes defined over larger fields tend to perform better than similar codes taken over $GF(2)$. In fact, as q increases, the performance improves. MacKay showed in [4] a regular LDPC code defined over $GF(16)$ with block length on the order of $24k$ bits that performs better by about 0.6dB than a comparable code over $GF(2)$. The same trend is seen when dealing with irregular LDPC codes. MacKay's results show an improvement of about 0.5dB when taking a code over $GF(8)$ instead of $GF(2)$.

Decoding complexity is an issue with larger fields. The decoding complexity for the sum-product algorithm as used by MacKay for his tests scales as q^2, so decoding over $GF(8)$ instead of $GF(2)$ results in a complexity increase by a factor of 16. There is work being done by MacKay and others in an attempt to find simpler methods to decode over larger fields in an optimal or near-optimal manner [4]. If the complexity can be reduced, even in a sub-optimal decoding scheme, the resulting performance improvement due to the larger fields may be able to offset the loss in performance in a sub-optimal decoder for close to the same amount of computational complexity.

One other area of interest for LDPC codes is their use in high rate or short block length applications, and this too has been examined by MacKay and Davey [15]. Their paper shows

the performance of a rate 8/9 code of short block length $n = 1998$ bits to be better than either comparable BCH codes or Reed-Solomon codes using hard-decision decoding by more than 1dB.

Despite their relatively poor performance with short block lengths, LDPC codes have been proposed as an FEC scheme for CDMA applications. Work on short block length codes by Sorokine [23] showed a class of irregular LDPC codes of block lengths on the order of 1000 and less to be particularly effective in fading channels. These codes appear to be well suited for carrying voice information on cellular networks, and allow for a greater user capacity per cell.

1.4 Organization of this work

The aim of this work is to investigate reduced complexity decoding methods for LDPC codes. Chapter 2 introduces the concepts and principles of LDPC codes, and outlines some of the algorithms that may be used for decoding. Density evolution [19], a technique which allows a decoding algorithm's performance to be determined numerically and aids in an overall understanding of the iterative decoding process, is also described in Chapter 2. Density evolution is applied in Chapter 3 to optimize reduced-complexity LDPC decoders. Trade-offs between performance and complexity are examined and presented. In Chapter 4, an alternate decoding approach is developed that exploits the duality of the LDPC factor graph and is well suited to fixed-point applications. Performance and complexity issues are treated, and possible implementation issues are discussed. Finally, Chapter 5 provides some conclusions and suggestions for further research.

Chapter 2

LDPC Code Fundamentals

The defining feature of LDPC codes is their low weight parity check matrices. It is important to understand the properties that this gives LDPC codes, and we will examine some here.

2.1 Construction of LDPC Codes

There are numerous methods for generating parity-check matrices that satisfy LDPC constraints, and a few commonly used techniques will be outlined here. While either a structured or a pseudo-random approach may be used for construction, it has been found that a higher degree of randomness generally results in codes that perform better than those that are highly structured.

Gallager originally proposed a method of partitioning the parity check matrix $\mathbf{H}$ into blocks and performing random permutation operations. Set the first row of $\mathbf{H}$ to d_c ones followed by $(n - d_c)$ zeroes [6]. Create the next $(n/d_c) - 1$ rows (assuming this is an integer) by shifting the previous row by d_c positions. This sub-section of the matrix is copied and randomly permuted a number of times to obtain the desired dimension for $\mathbf{H}$. An example of this construction is given in (2.1). This construction generates regular codes, and is sure to have low probability of repeated rows. However, since the sum of the rows of the top block and the rows of any other block is zero, this method results in linearly dependent rows and as such, the rate of the code is higher than might first be assumed.

$$H = \left[\begin{array}{cccccccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{array}\right] \begin{array}{l} \left.\begin{array}{l} \\ \\ \end{array}\right\} \text{Top Block} \\ \left.\begin{array}{l} \\ \\ \end{array}\right\} \text{Permutation of Top Block} \\ \left.\begin{array}{l} \\ \\ \end{array}\right\} \text{Another Permutation of Top Block} \end{array} \tag{2.1}$$

An alternate method was presented by MacKay that allows a nearly uniform parity check matrix to be generated [16], with no strict guarantee of uniformity. The construction operates by assigning a fixed number of ones, j, to each column of the parity check matrix. A bit in a particular column is assigned to a row with uniform probability, with no regard to whether that particular location is already occupied by a one. It is possible then to have a column with lower weight than j, though it is not possible to have a column with greater weight, resulting in an irregular code. MacKay has also suggested that any rows which share multiple ones in the same location should be avoided. This prevents short cycles from appearing in the graph, which has been shown to reduce the performance of LDPC codes.

A more general method of LDPC parity check matrix construction is simply to assign one's randomly with low probability to a matrix of the desired dimensions. This will almost surely result in an irregular code, with average row and column weights that can easily be fixed. For example, the matrix

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{2.2}$$

was obtained in this way. Care must be taken to ensure that there are no identical rows, and that linear dependence among the rows is kept relatively small.

Although the matrix in (2.2) is not likely to be considered sparse, it is a good demonstration of the generation of an irregular LDPC parity check matrix. The polynomials describing the distribution of the degree's of the check and variable nodes are given in (1.3), and the bipartite graph representing this code is shown in Figure 1.1.

MacKay has also outlined a modification to his construction technique that allows the resulting LDPC code to be encoded much more efficiently, based on the work of Spielman [16, 13, 24]. This will be discussed in detail in Section 2.2.3.

2.2 Properties of Low Density Parity Check Codes

It is useful to first outline some terminology. In defining the ability of a family of codes to achieve capacity, at least asymptotically, the terms *very good*, *good* and *bad* are used by MacKay to describe this attribute and are adopted here [14]. Very good codes are those that can asymptotically achieve an arbitrarily low probability of bit error for any rate below capacity using some decoding algorithm (e.g., maximum likelihood decoding). Good codes are codes that can achieve this same performance, but only up to some rate which is bounded away from the channel capacity. A bad code is any code that has some negligible probability of bit error only for a vanishingly small rate. It is also useful to define a *practical* code as one which can be implemented effectively with limited computational complexity, i.e., encoded and decoded in a useful amount of time, depending on the application.

2.2.1 Goodness properties

In Gallager's original paper on the topic, he proves that LDPC codes are good with a simple decoder [6]. This is based on a binary symmetric channel (BSC), and uses a bit-flipping iterative decoding algorithm he devised which will be discussed in Section 2.2.4. The LDPC code is limited to achieving arbitrarily low probability of bit error only for channel errors occurring with a frequency less than that which corresponds to capacity of the BSC.

MacKay shows in [14] that not only are these codes good, but they are very good when certain criteria are met using a sub-optimal typical set decoder. First define a satisfactory matrix as:

> A satisfactory $(M, \lambda, t, \epsilon)$ matrix $\mathbf{A}$ for the distribution $P(\mathbf{x})$ is a matrix having M rows and $L \geq \lambda M$ columns with weight t or less per column, with the following property: if $\mathbf{x}$ is generated from $P(\mathbf{x})$, the optimal decoder achieves a probability of bit error less then ϵ. [14]

MacKay's theorem states that given some distribution $P(\mathbf{x})$ of the probabilities of inputs to the decoder with mean entropy $H(X) < 1$ bits, and a desired code rate $\lambda < 1/H(X)$, there exists some integer $t(H(X), R) \geq 3$ with the following implication: There exists some row dimension of $\mathbf{H}$ above which there is a satisfactory $(M, \lambda, t, \epsilon)$ matrix $\mathbf{A}$ that can achieve a block error probability of $\epsilon > 0$. In this case, t represents the column weight of $\mathbf{H}$, or d_v. Thus as d_v is allowed to grow large, capacity may be achieved. Unfortunately, the complexity of most LDPC decoding algorithms grows as d_v gets large.

2.2.2 Distance Properties

Regular LDPC codes have the strict constraint on their parity check matrix that each row have some fixed weight and each column also have some fixed weight. With just these constraints, Gallager was able to give bounds on the performance of this ensemble of random codes, bound the distance functions, and provide a powerful yet simple decoding technique, which can be generalized to belief propagation.

The LDPC codes Gallager examined exhibit good distance properties for $d_v > 2$, and

he showed that the minimum distance of these codes grows linearly with block length n for most codes in the ensemble. MacKay showed in [14] that the Gilbert-Varshamov minimum distance bound can be obtained as $d_v \to \infty$. For a parity check matrix $\mathbf{H}$ with parameters d_v and d_c as defined above, the Shannon limiting bound on entropy for the codeword $\mathbf{x}$ is $H(\mathbf{x}) \leq n/m$ where m is the size of the information block. It can then be said that for $d_v > 3$ and a fraction $\delta < 0.5$, then some $\mathbf{H}$ can be found such that the minimum distance of the code is at least δn.

2.2.3 Encoding

One of the obstacles preventing LDPC codes from being widely adopted in practical applications is the complexity involved in the encoding process which can be many times greater than other codes such as convolutional codes or turbo codes. A great deal of attention has been paid to the performance of LDPC codes and efficient optimal decoding methods, but the encoding problem has often been overlooked. Using standard matrix multiplication to generate codewords $\mathbf{c}$ from a generator matrix and message word $\mathbf{m}$, i.e. $\mathbf{c} = \mathbf{m}G$, encoding complexity increases to the second power of the block length n, where one would hope that there would be a linear relationship between encoding complexity with block length.

Spielman was one of the first to examine how sparse matrix techniques may be used to efficiently encode LDPC codes [24]. His codes, which are a subset of codes called *expander codes*, are linear time encodable. These codes however do not perform as well as other LDPC codes and are not good codes in the technical sense.

MacKay took the work of Spielman in [25] and developed a nearly lower-triangle reduced matrix form for $\mathbf{H}$ that is almost linear time encodable as mentioned above in section 2.1 [16, 15]. The structure of $\mathbf{H}$ is lower-triangular except for the last $M_<$ rows, which allow all but these last $M_<$ rows to be encoded using sparse matrix techniques, which is a linear time effort. It is not possible to express an LDPC code fully in lower-triangular form as this would violate the LDPC constraints, but it can be made very close. In fact, if the total number of rows in $\mathbf{H}$ is M, and if $M_< \leq \sqrt{M}$ then MacKay has suggested that the total code can be linear time encodable, though he has not expressly proven that this is possible.

2.2.4 Decoding

An LDPC decoder employs *iterative* decoding, and its goal is to compute the *a posteriori probability* (APP) of each of the received symbols, taking the symbol with maximum probability as the decision. There are many ways this can be done, one of which is the *sum-product* algorithm, a practical decoding algorithm. This approach will compute the exact APP for each symbol in a cycle-free graph, and in the case of a graph with cycles, will generally compute close approximations of the APP though this behaviour is not clearly understood. We will examine a basic instantiation of the MAP decoder to explore its properties.

Gallager devised an algorithm that is similar to sum-product decoding and developed an analysis of iterative decoding for the binary symmetric channel (BSC) using this algorithm. This attempt at a clear analysis of iterative decoding is the first step towards a deeper understanding of this decoding approach. This is the first step towards the concept of *density evolution*, a powerful analytical technique for iterative decoding.

A basic bit-flipping algorithm known as Gallager's Decoding algorithm A [6] is described here, and other algorithms will be examined in Chapter 3. The procedure is a graph based iterative decoding method using message passing as described in Section 2.3, and assumes a BSC. Bits are read off the binary channel, and are loaded into the variable nodes of the graph. These bits are the messages which are passed along the edges of the graph and are used for decoding. Messages are successively passed from variable nodes to check nodes, and from check nodes to variables, with new messages being computed at the nodes.

At a check node, parity is computed as follows. Each edge incident on a given check node computes and returns along that edge, parity using modulo-2 addition of all the bits incident on the check, excluding the edge for which parity is being computed. These results are passed back along the edges to the variable bits. At a variable node, the value of the bit sent out along an edge is flipped only if *all* of the other edges incident on that node return equal and opposite values from the current value. So a 1 along an edge will be flipped to a 0 only if all of the other edges incident on that node return the value 0. The procedure then repeats until some termination criterion is satisfied.

Given that the received bits are 1's and 0's, let p_i be the probability that a bit is in error

after the i^{th} round of decoding. For some bit in error, a parity check involved with that bit will be incorrect only if an even number of other bits in the check is in error. An error will be corrected only if all of the checks participating with that bit are not satisfied, and this occurs with probability:

$$\left[\frac{1+(1-2p_{i-1})^{d_c-1}}{2}\right]^{d_v} \tag{2.3}$$

Similarly, it is possible for a correct bit to be mistakingly flipped to an incorrect state, and that probability is:

$$\left[\frac{1-(1-2p_{i-1})^{d_c-1}}{2}\right]^{d_v} \tag{2.4}$$

Applying these results, a recursive relation can be found for the next round's overall probability of bit error:

$$p_i = p_0 - p_0\left[\frac{1+(1-2p_{i-1})^{d_c-1}}{2}\right]^{d_v-1} + (1-p_0)\left[\frac{1-(1-2p_{i-1})^{d_c-1}}{2}\right]^{d_v-1} \tag{2.5}$$

This result is an important step towards an understanding of iterative decoding. Gallager showed that there exists some maximum value p_0^* of p_0 for which p_i goes to zero as i gets large for all p_0 less than p_0^*. In other words, if the initial probability of error created by the channel is below some threshold, there is a very high probability for the received word to be properly decoded. This shows that even for the simple decoding algorithm that Gallager used, LDPC codes can achieve an arbitrarily low probability of bit error for some rate that is bounded away from capacity.

Clearly, there are more powerful ways to decode LDPC codes. Gallager's Algorithm A does not use soft-decisions, and assumes a BSC channel. We shall now explore message passing on graphs to see how more powerful algorithms may be exploited.

2.3 Factor Graphs and Message-Passing Algorithms

Factor graphs are bipartite graphs that describe how a function of many variables is comprised of its constituent factors [10]. Figure 1.1, showing the random LDPC code construction in a graphical form, is an example of a factor graph. In graphical notation, variables are shown as circles, and function nodes are squares, sometimes with an operator in them to

describe the function that is being performed at that node. For example, the factor graphs for LDPC codes often have a '+' sign representing the constraint that neighbouring variables have zero sum modulo-2.

We can use this representation to provide a framework for algorithm execution, and in this case we are interested in investigating decoding algorithms. To assist in understanding message passing, an example for the $(7,4)$ Hamming code will be investigated. A parity check matrix for the Hamming code is given by

$$\mathbf{H} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{2.6}$$

As a linear block code, we are interested in finding the a-posteriori probabilities (APP) for each received symbol

$$f(x_i \mid \mathbf{y}) = \sum_{x:x \neq x_i} f(x_1, x_2, x_3, x_4, x_5, x_6, x_7 \mid \mathbf{y}) \tag{2.7}$$

where i is the i^{th} digit in the code and the function $f(\cdot)$ represents the conditional joint probability mass of $(x_1, \cdots, x_7)$ given the received channel output $\mathbf{y}$. The goal is to calculate each of these probabilities based on the constraints of the code. In order to properly see this, the first digit x_1 will be examined, noting that the analysis proceeds similarly for all other bits.

We know from Bayes' rule that the probability of having sent a codeword $\mathbf{x}$ upon receiving the word $\mathbf{y}$ from the noisy channel is

$$f(\mathbf{x} \mid \mathbf{y}) = \frac{f(\mathbf{y} \mid \mathbf{x}) p(\mathbf{x})}{f(\mathbf{y})} \tag{2.8}$$

where $f(\mathbf{y})$ can be viewed as a constant since $\mathbf{y}$ is known upon reception of the word, and $p(\mathbf{x})$ depends on the code used. If we assume independence of the channel symbols, i.e. a memoryless channel, and assuming that codewords are selected with uniform probability, then we can factor (2.8) into

$$f(\mathbf{x} \mid \mathbf{y}) = K(\mathbf{y})[\mathbf{x} \in C] \prod_i f(y_i \mid x_i) \tag{2.9}$$

where K is a constant factor that depends only on $\mathbf{y}$, and the brackets $[\cdots]$ follow Iverson's convention[1]. The above probability can be further factored into each of the constraints of the code,

$$f(\mathbf{x} \mid \mathbf{y}) = K \prod_s [\mathbf{x}_s \in C] \prod_i f(y_i \mid x_i) \tag{2.10}$$

For the example of the Hamming code, this factorization can be expressed as

$$f(\mathbf{x} \mid \mathbf{y}) = K[x_4 \oplus x_5 \oplus x_6 \oplus x_7 = 0][x_2 \oplus x_3 \oplus x_6 \oplus x_7 = 0][x_1 \oplus x_3 \oplus x_5 \oplus x_7 = 0] \prod_i f(y_i \mid x_i) \tag{2.11}$$

where the three boolean propositions are derived from the three rows of (2.6) representing the constraints of the code. Since we are interested in probabilities for each symbol, we compute marginals for x_i, which for the case of x_1 is

$$\begin{aligned} f(x_1 \mid \mathbf{y}) = \; & f(x_1 \mid y_1) \sum_{x_7} f(x_7 \mid y_7) \sum_{x_5} f(x_5 \mid y_5) \sum_{x_3} f(x_3 \mid y_3)[x_1 \oplus x_3 \oplus x_5 \oplus x_7 = 0] \cdot \\ & \sum_{x_6} f(x_6 \mid y_6) \sum_{x_2} f(x_2 \mid y_2)[x_2 \oplus x_3 \oplus x_6 \oplus x_7 = 0] \sum_{x_4} f(x_4 \mid y_4)[x_4 \oplus x_5 \oplus x_6 \oplus x_7 = 0] \end{aligned} \tag{2.12}$$

We see then that the problem of computing the marginals has been factored into many smaller problems, and this is naturally expressed through the use of factor graphs and a message passing algorithm that passes these marginals as messages. Unfortunately this factor graph has cycles, so the marginals computed are not exact. This example is used however, since virtually all codes have factor graph representations with cycles.

The goal of MAP decoding is to find the maximal a posteriori symbols of a noise corrupted codeword. In general, this can be factored into the problem of finding the most likely symbol x_i given that the symbol y_i has been received and that the transmitted word $\mathbf{x}$ is subject to the constraints of the code. The sum-product algorithm applied to the factor graph for the code of interest calculates the most likely transmitted symbols in this manner.

Message passing on a factor graph describes how the calculated probabilities on the graph will behave over time given a certain set of initial conditions. There are numerous algorithms

[1]Iverson's convention takes a boolean proposition and returns a '1' if that proposition is true, otherwise a '0' is returned. For example, given a proposition P, $[P] = \begin{cases} 1 \text{ if P} \\ 0 \text{ otherwise} \end{cases}$.

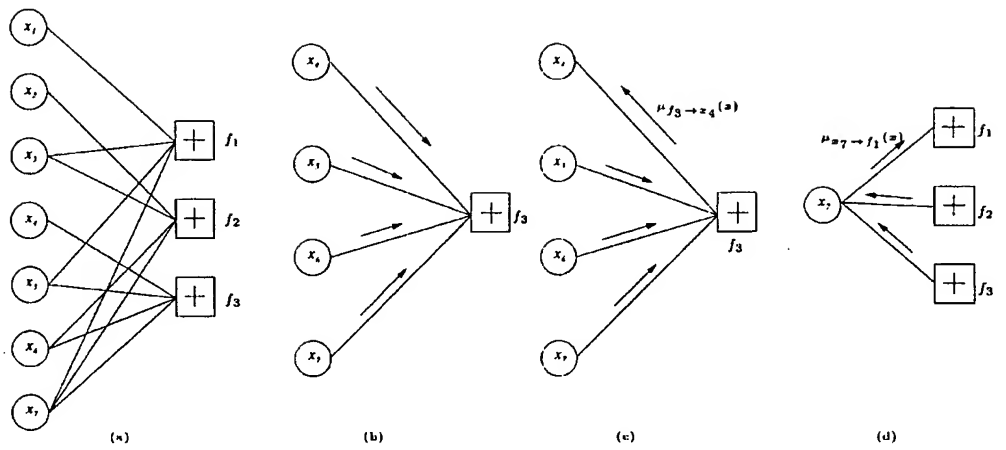

Figure 2.1: (a)Factor graph representation of a (7,4) Hamming Code. (b) Initial message passing to a check node. (c) Return message from a check node. (d) Return message from a variable node.

that can be used to perform message passing, but in particular the sum-product algorithm will be described as it can be used to decode LDPC codes quite powerfully in many cases. The BCJR algorithm is actually a specific implementation of the sum-product algorithm, and belief propagation and turbo-decoding also turn out to be variations of the sum-product algorithm. These types of message passing are *iterative* decoding algorithms.

The message passing algorithm commences by initializing each of the variable nodes with the corresponding received symbol probability. Some message passing schedule is used which sequentially passes all the information from each node to each adjacent node. There are many schedules which can be used, and it is up to the decoder designer to determine a suitable one. For instance, a sequential schedule in which each message is updated in turn such that no message is updated twice before any other message is updated may be used. This schedule is used in all trials performed in this work. The effect of varying message schedule may have some impact on the performance of the decoder, but the schedule chosen was found to result in good performance.

Messages are represented by the symbol μ, and represent some measure of the *reliability* of the information being passed. The rule determining the update of the information at each

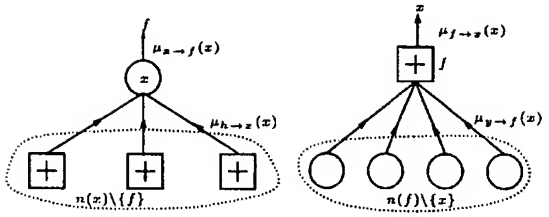

Figure 2.2: Generalized message passing at variable and check nodes

node is:

> *Sum-Product Update Rule* [10]: The message sent from a node v on an edge e is the product of the local function at v (or the unit function if v is a variable node) with all messages received at v on edges *other* than e, summarized for the variable associated with e.

The summarization operator refers to the computation of a marginal, meaning that the value is all possible configurations for all values *excluding* the one which is of interest. Figure 2.2 provides more detail. Within each node, it is assumed that calculations are performed using an efficient method, so in all cases the forward-backward algorithm is used [11], as described in Appendix B.

There are many probability metrics that may be used for messages, and a few are outlined in Table 2.1. Depending on what what domains are chosen for representing messages, the form that the sum-product update rules take may vary drastically. Two representations of particular interest are the log likelihood ratio (LLR) and the log likelihood difference pair (LLDP). Using log likelihood ratios, we can write the variable node update rule as

$$\mu_{x\to f}(x) = \mu_x^{ch} + \sum_{h\in n(x)\setminus f} \mu_{h\to x}(x) \tag{2.13}$$

where x is the variable node of interest, f is the check node whose edge is attached to x along which we are sending the output. The value μ_x^{ch} is the message from the channel corresponding to the variable node x being updated, and $n(x)$ is the set of edges connecting

Name	Representation (μ)	Domain ($\in \mathbb{R}^{\infty}$)
Basic	$p[0], p[1] = 1 - p[0]$	$[0, 1]$
Likelihood Ratio	$\frac{p[0]}{p[1]}$	$[0, \infty]$
Likelihood Difference	$(p[0] - p[1])$	$[-1, 1]$
Log-likelihood Ratio(LLR)	$\log\left(\frac{p[0]}{p[1]}\right)$	$[-\infty, \infty]$
Log-likelihood Difference pair(LLDP)	$(\text{sgn}(p[0] - p[1]), \lvert\log\lvert p[0] - p[1]\rvert\rvert)$	$(\{\pm 1\}, [0, \infty])$

Table 2.1: Possible binary message representations for iterative decoding algorithms

to x. The check node update equation using log-likelihood ratios is

$$\mu_{f\to x}(x) = \log\left(\frac{\prod_{y\in n(f)\setminus x}(e^{\mu_{y\to f}(x)}+1) + \prod_{y\in n(f)\setminus x}(e^{\mu_{y\to f}(x)}-1)}{\prod_{y\in n(f)\setminus x}(e^{\mu_{y\to f}(y)}+1) - \prod_{y\in n(f)\setminus x}(e^{\mu_{y\to f}(z)}-1)}\right) \tag{2.14}$$

where d_c is the number of edges connected to f, the check node being updated. This rule is actually quite complicated, and is one of the drawbacks of the sum-product algorithm with this representation. It is interesting to note that the update rule for the check nodes using the log-likelihood difference pair is very similar to (2.13), and may be written as

$$\mu_{x\to f}(s, r) = \sum_{h\in n(x)\setminus f} \mu_{h\to x}(s, r) \tag{2.15}$$

where $s \in W_2 \equiv \pm 1$, $r \in \mathbb{R}^+$ and the addition of the signs in $\{\pm 1\}$ is real multiplication. Later, we will exploit this representation in an attempt to reduce the decoding complexity.

Thus, returning to the example of the $(7, 4)$ Hamming code, we can follow the algorithm through. As mentioned previously, the seven variable nodes are initialized to be the corresponding probabilities representing the likelihood of the received bits being either a 0 or 1. Each of the seven variable nodes sends its likelihood message along each edge to the corresponding check nodes. The message that is returned to each of the variables is the result of the check node function with all message inputs excluding the input message along the edge that the output message is passed. Figure 2.3 (c) depicts the return message to x_4 based on the inputs of x_5, x_6, and x_7 to the parity check node. Each variable node is updated

following the same rule, such that the output along an edge is dependent on all of the inputs to the node excluding the one along the output edge.

At this stage, one full iteration through the graph has been completed and all of the variable nodes have an updated estimate as to their probability mass function. This iterative procedure continues until some termination criterion is met. This could possibly be a convergence measure on the variable node output messages, a threshold on the maximum number of iterations, or a check node 'satisfaction'. Once the algorithm is complete, a decision is made based on each bit's probability mass whether it is more likely a 0 or a 1 and this hard decision is output as the decoded word. It is also possible to output *soft* decisions, meaning that the reliability measure is also output along with the decision.

This approach provides a relatively straightforward method to decode LDPC codes. One problem that has not been mentioned is the case of cycles in graphs, which are unavoidable in LDPC codes and most other codes of finite length. For the cycle free case, the sum-product algorithm exactly computes the desired factors, but this is not the case for graphs with cycles. Despite this drawback, the sum-product algorithm still works extremely well on graphs with cycles, as the APP estimates are 'good enough'. Many different simulations have shown that for codes without small cycles (of length$\leq$ 2), the sum-product algorithm works very well.

2.4 Density Evolution

Since the rediscovery of LDPC codes in the mid 1990's, a great deal of work has been done to further develop the performance and understanding of LDPC codes and iterative decoding schemes in general. A first extension of Gallager's analysis for decoding outlined in Section 2.2.4 was presented by Luby, Spielman, et al., who used the same bit-flipping algorithm applied to irregular LDPC codes [12].

In late 1998, Richardson and Urbanke applied the bit-flipping algorithm to a number of channels including the Gaussian channel, and found the maximum allowable parameters for these channels, verifying Gallager's results and presenting new ones [19]. They were also able to successfully analyze the capacity of LDPC codes using soft-message-passing algorithms,

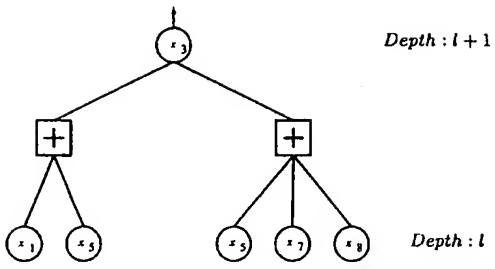

Figure 2.3: Tree representation of a section of figure 1.1

one of which is the sum-product algorithm described in Section 2.3. Their approach will be detailed here, though expanded to include irregular codes.

First it is useful to describe how a factor graph can be *unvolved.* Message passing algorithms work on graphs and it is insightful to be able to view how the messages move through the graph as a function of time. Time in this case is assumed to be some discrete counter that advances at a rate that is a function of the message passing algorithm. For example one iteration of a decoding algorithm may be considered to be one time step. It is possible to move backward in time from a certain node n in the graph and recursively view all of the nodes and edges which have contributed to the message being output from n.

If we draw the node n and all of the connected edges and nodes which participate with node n back t time steps, we have an unvolved tree as in Figure 2.3. This is an unvolved graph of depth 2. The resulting graph is a *tree* or is *tree-like*, if and only if all of the nodes in the tree are distinct. This is not the case in Figure 2.3 as the variable x_5 is repeated, but it serves its purpose as an example. A message passing algorithm used on the graph sends messages up the tree, and we are able to understand how the distribution of messages and each level l in the tree affects the message output at depth $l + 1$. Thus we see that this represents the time evolution of messages on the graph. We would like to find out how the distribution of the messages evolve as they proceed up the tree in order to understand how iterative decoding methods work. First, some assumptions need to be made about the system.

The all zeros' codeword is to be the transmitted codeword. If symmetry conditions

of the channel and both variable and check nodes are satisfied, then the probability of a decoding error is independent of the transmitted codeword, validating the use of this word and is proved in [19]. The all zeros' codeword also has the advantage of having a simple distribution of bits, namely a delta function, and this will assist in the simplicity of the analysis.

The symmetry conditions that are needed for the use of the all zeroes word are symmetry in the channel, and symmetry at both check and variable node update functions. Symmetry in the channel implies zero-mean noise for the AWGN or an equal likelihood of flipping a bit on the BSC. Symmetry at the nodes means that given a set of input message reliabilities, i.e. message magnitudes, to the function, the output reliability should be independent of the signs of the inputs. What these conditions impose is a prevention from the possibility of distributions erroneously drifting.

We will assume that the channel is memoryless and all received symbols are *i.i.d.*. Each of these symbols is converted to a probability metric and is used to initialize the 0^{th} level of the tree. The distribution of these probabilities can be denoted as P_0. Working under the assumption that the code is tree-like to a depth l, where l is the decoding round, we can develop an exact analysis of the densities at the l^{th} level of the tree.

Let the messages sent from variable nodes to check nodes be represented by LLRs, and let the messages sent from check nodes to variable nodes be represented by LLDPs. Let P_l be the probability distribution (density) of messages sent from the set of all variable nodes, and let R_l be the density of messages passed from the set of all check nodes, both at round l in the decoding process (or depth l on the tree). Also, define

$$\tilde{\lambda}(f) \equiv \sum_i \lambda_i f^{\otimes(i-1)} \text{ and } \tilde{\rho}(f) \equiv \sum_i \rho_i f^{\otimes(i-1)} \tag{2.16}$$

where f is an input distribution, and the notation $f^{\otimes x}$ represents a repeated convolution of the distribution f, x times. Both λ_i and ρ_i represent the same degree sequence statistics as in (1.2). Using the LLR representation as messages, we have an expression for the output of a variable node given in (2.13). The density of these messages for all variable nodes is the convolution of all the input densities participating. This can be written as

$$P_l = P_0 \otimes \tilde{\lambda}(R_{l-1}) \tag{2.17}$$

where $\otimes$ denotes convolution. Similarly, representing the messages being passed to a check node as an LLDP, (2.15) can be used to determine the output message. Again, the density of these messages is the convolution of all of the participating input densities, and can be written

$$R_{l-1} = \tilde{\rho}(P_{l-1}) \tag{2.18}$$

In order to develop a relationship between the densities from one round of decoding to the next, we require a transformation function which can convert densities from the LLR domain to the LLDP domain and back, which are derived in [18] and given here. To change a probability distribution from either representation to the other one may apply the following change of variable

$$y \rightarrow \ln\coth y/2 \quad \forall y \geq 0 \tag{2.19}$$

which, for probability distributions, may be written as the functional

$$[\gamma f](y) \equiv \frac{f(\ln\coth y/2)}{\sinh(y)} \quad \forall y \geq 0 \tag{2.20}$$

This operator maps distributions over the reals, $f(y)$ to distributions over the reals:

$$\gamma : \mathbb{R}^{\mathbb{R}} \rightarrow \mathbb{R}^{\mathbb{R}} \tag{2.21}$$

Note that

$$\ln\coth\left(\frac{\ln\coth y/2}{2}\right) = y \tag{2.22}$$

so it follows that $[\gamma[\gamma f]](y) = f(y)$. Thus we have a method of converting positive distributions of probabilities from one domain to the other.

For some density g, let $g = g^+ + g^-$, where g^+ and g^- represent the positive and negative portions of the density respectively defined by

$$\begin{aligned} g^+(x) &= g(x), x \leq 0 \\ g^+(x) &= g(x), x \geq 0. \end{aligned} \tag{2.23}$$

Also define a reflection in the y-axis operator, $\mathcal{R}$, defined as $[\mathcal{R}g^-](x) = g^-(-x)$. Finally define the change of variable operator that takes as it's input an LLR distribution over $\mathbb{R}$ and returns an LLDP distribution over $W_2 \times \mathbb{R}^+$

$$\Gamma : \mathbb{R}^{\mathbb{R}} \rightarrow \mathbb{R}^{W_2 \times \mathbb{R}^+} \tag{2.24}$$

where W_2 is the set $\{\pm 1\}$. This operator can then be defined as

$$[\Gamma f](s,r) = (1, [\gamma f^+](r)) + (-1, [\gamma[\mathcal{R} f^-]](r)) \tag{2.25}$$

and it's inverse, which takes an LLDP input distribution and returns the equivalent LLR distribution is defined as

$$[\Gamma^{-1} f](x) = [\gamma f^+](x) + [\mathcal{R}[\gamma f^-]](x) \tag{2.26}$$

Given the previous equations and operators, we can combine them to produce the heart of Richardson and Urbanke's work, their theorem of density evolution on continuous message passing algorithms. If the distribution of messages from check nodes is initialized to erasures $R_0 = \delta(0)$, i.e. $p[0] = p[1]$ for all messages, then we can write the theorem of density evolution as:

> Density Evolution [18]: *Let P_0 denote the initial message distribution, under the assumption that the all zeros codeword was transmitted, of an LDPC code specified by edge degree sequences $\lambda(x)$ and $\rho(x)$. If R_l is the distribution of messages passed from check nodes to message nodes at round l of sum-product decoding, then*
>
> $$R_l = \Gamma^{-1} \tilde{\rho}(\Gamma(P_0 \otimes \tilde{\lambda}(R_{l-1}))) \tag{2.27}$$

This powerful result provides a method to find good LDPC codes that perform very well without the need of simulating codes on a trial and error basis. Degree sequence polynomials $\rho(x)$ and $\lambda(x)$ can be found which maximize a particular channel parameter, such as the threshold SNR, thus providing near-optimal performance. Since the all zeros' codeword is sent, we know that the algorithm has converged to the correct codeword when the density approaches a delta function where $p[0] = 1$, which is at positive infinity for the LLR representation.

It should be noted that we cannot actually converge to a delta function at infinity, but it can be approached. Thus, the resulting LDPC codes used with the optimized degree sequences may not asymptotically perform as well as expected since the probability of bit

error is not zero. It is vanishingly small however, and empirical evidence supports the validity of asymptotic performance.

Hence, we can try to find degree sequences using some directed search that maximizes P_0 for which the density evolution converges to the all zero codeword distribution, and this maximum is labelled as P_0^*. This threshold is the supremum of all channel parameters for which the probability of error approaches zero for some number of iterations on the graph. For example, in the AWGN the noise variance σ can be maximized for which the BER of the decoder is vanishingly small.

One beneficial feature that stems from this form of analysis is that an additional constraint on the number of iterations used to decode can be enforced. Since level l of the tree corresponds to the l^{th} decoding iteration, we can attempt to find degree sequence polynomials for which P_0^* is maximized within a constrained number of iterations. This is very useful when searching for good codes that are also practical.

Using these density evolution methods described, Richardson and Urbanke were able to develop what they term as *highly irregular LDPC codes* that asymptotically perform to within 0.06 dB of the Shannon limit [18]. This is a significant increase over the previous best known performing code. In the subsequent chapters, we make use of this approach to optimize LDPC codes for reduced complexity decoding algorithms.

2.4.1 Fixed Points

It is useful to examine how the the distribution of messages passed during the decoding process evolves over time. To this end we define a statistic P_l^E which represents the proportion of messages being passed at round l that, if a hard decision was made as to their sign, are in error.

Using this statistic, the time evolution of errors through the message passing algorithm may be plotted, and two examples are given in Figure 2.4. The curve to the right represents how the distribution of messages approaches the correct codeword ($P_\infty^E = 0$) as the decoding round gets very large, in this case about 1600. The inner curve gives the same evolution for density evolution constrained to maximum decoding iterations of less than 200. Note that

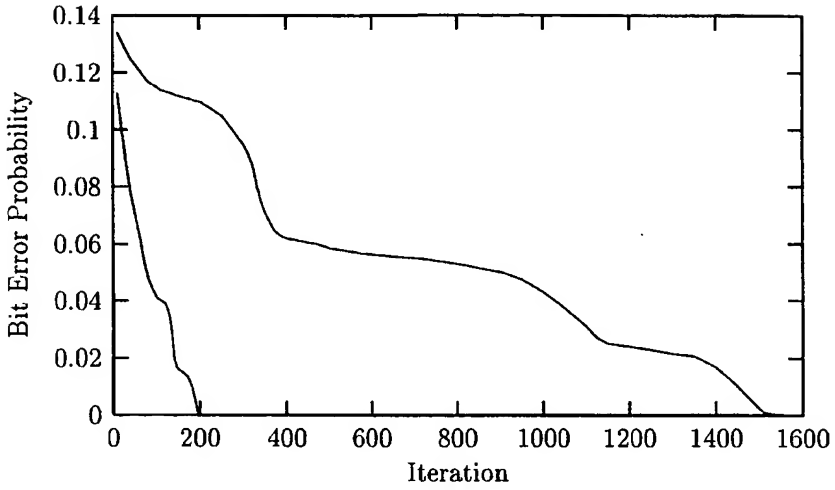

Figure 2.4: Time evolution of hard-decision errors in message-passing. The outer curve represents evolution of a code optimized with no iteration constraints, the inner curve code was optimized with a constraint to a maximum of 200 iterations.

the curves are monotonically decreasing but their slope fluctuates. The regions where the slope approaches 0 are called *fixed points*, and are locally near-stable regions of the graph. At these fixed points, the incremental improvement in P^E is very small such that for realizable codes, i.e., codes of finite length with cycles, it is possible that the evolution will become 'stuck' at these points.

In fact, this is the most common cause of decoder failure. Recall that density evolution allows us to find highly optimized LDPC codes and these codes are very sensitive to any deviations in their structure. Codes of shorter block length tend to deviate further from the optimal degree sequences due to finite effects. These deviations result in the possibility of the slope at a fixed point to actually reach 0 depending on the actual received probabilities, such that the code will not improve any further, and a decoding error results. Techniques to mitigate this effect are discussed in [18].

Chapter 3

Optimization of LDPC Decoders

In Chapter 2, we developed the concepts of density evolution, and presented a relationship describing the evolution of densities over time though the unvolved tree. We will expand on these ideas in this chapter and develop a similar approach to analyzing decoding algorithms that have a reduced complexity compared to the sum-product algorithm.

The ability to find the threshold performance for a LDPC code leads to the problem of how one can find good degree sequence polynomials that maximize these thresholds. This becomes a non-linear optimization problem with a huge parameter space. We can utilize a powerful, global optimization technique called *genetic algorithms* to provide the means to solve this problem.

3.1 Genetic Algorithms

A genetic algorithm (GA) is a global optimization technique which takes the concepts of biological evolution and Darwinian survival and applies them to finding optimal solutions to some generally hard problem. The basic operation of a GA is conceptually simple: One creates a population of solutions to some problem, selects and combines the better solutions in some manner, and uses their recombination to replace poorer solutions. The combination of selection pressure and innovation generally leads to improved solutions, often approaching the globally optimal solution.

The problem to be solved must be expressed through artificial *chromosomes*, each of which represents one possible solution to the problem. A chromosome may be a string of bits, a list of parameters, or a more complex representation, such as a tree of procedures, but it must represent some embodiment of a solution to the problem. For example, suppose the problem is to find the point furthest away from the origin that is within the bounds of some closed region of a 2-dimensional space. One simple method to encode the problem is to use a chromosome that contains two real numbers (x, y) representing a location on the 2-D surface.

The GA requires a method to evaluate how good or bad a particular solution, or chromosome, is. This is usually some function that takes as its inputs the chromosome, and outputs a *fitness* measure, and the fitness is what the GA uses to score the potential solution. Using our previous example, the GA's fitness function may take the chromosome, which is an (x, y) coordinate, measure its Euclidean distance from the origin, and determine whether or not is is within the bound of the closed function in question. If it is, it may return a fitness equal to the distance from the origin, otherwise it may return some non-valid measure, such as -1.

Having encoded the problem onto a chromosome, and having an evaluation procedure, the GA will *evolve* solutions to the problem by creating a *population* of chromosomes. A population is generally randomly created, and for each individual in the population, its fitness is determined. The GA will then select those individuals with the highest fitness, and perform some *genetic* operations on them to create a new generation of individuals for the population, which replace the individuals in the population with generally the worst fitness. The effect, hopefully, is an overall improvement in fitness.

Genetic operators include *selection*, *recombination*, and *mutation*, as well as other less common ones. Selection is the basic idea that a GA will prefer better solutions to worse solutions. This is Darwin's 'survival of the fittest' in action. There are many ways selection can be accomplished, from choosing the individuals with the highest fitness, to random chance. Recombination combines various parts of the chromosomes from two of more *parents* and produces a new chromosome *child* that contains traits from its parents, just as in biological systems. Mutation makes some sort of alteration to a chromosome, affecting some change in the trait of that chromosome, such that its fitness may be altered.

Using these operators, the GA repeatedly replaces members of the population and over a number of generations, the average fitness of the population will hopefully improve, as worse individuals are replaced. Usually, the population will converge to some steady-state distribution, meaning an optimum has been found. It may be a local optimum, not global, and it is often difficult to know which it is. GA practitioners make sure that the initial population is large enough and random enough that the entire search space is adequately covered, and they control the genetic operators in such a way to keep the rate of evolution from converging too quickly on a local optimum before a better optimum has been found.

For the work in this paper, the GA package GALib was used [27]. This source-code is a library of C++ classes that allows the user to easily set up chromosomes, fitness functions, and define genetic operators. It is available for use from MIT at the URL: `http://lancet.mit.edu/ga/` and contains a full description of how the package works.

3.1.1 Optimization Approach

The potential search space for this problem is enormous, being all possible degree sequences defining an LDPC code, and it is useful to constrain it in order to reduce the overall complexity. To this end, we constrain the rate of the code to be a fixed value, and for all of the cases presented, this value was $R = \frac{1}{2}$. We also constrain the maximum degree that any check or variable node can take. The goal of the optimization is to maximize the threshold performance of an LDPC code through the use of density evolution. As we see in [18], the threshold performance increases as the maximal degrees increase, so we should be aware of the impact on threshold performance by limiting the maximal degrees for $\lambda(x)$ and $\rho(x)$.

Let the maximal degree allowed for $\lambda(x)$ be d_v^*, and for $\rho(x)$ be d_c^*. Define the chromosome for the genetic algorithm to be a list of real numbers $q \in [0, 1]$ such that

$$\text{chromosome} \equiv q_2^v, q_3^v, \cdots, q_{d_v^*}^v, q_2^c, q_3^c, \cdots, q_{d_c^*-1}^c \tag{3.1}$$

where q_i^v and q_i^c correspond to the variable and check coefficients of degree i respectively. In order for the polynomials $\lambda(x)$ and $\rho(x)$ to meet their coefficient constraint that $\sum_i \lambda_i = 1$ and $\sum_i \rho_i = 1$, we normalize both sections of the chromosome in the fitness function. Note that we do not include an element in the chromosome for the check node of highest degree.

This is to allow the rate constraint to be met, as the check node variables are normalized so that the desired rate of the resulting code is achieved.

The initial population of size 50 individuals is randomly generated, and the GA commences evaluation and evolution. The fitness function takes as its input the chromosome, and converts it into the degree sequence polynomials such that the desired code rate is correct. The function performs density evolution to determine the threshold performance for the ensemble of codes described by the polynomials, and the threshold is returned by the function as the fitness score. The number of iterations allowed can also be fixed to limit the search to more practical codes. Trials performed had the maximum number of iterations limited to 25, 50, 100, and 200.

Those individuals with the highest fitness are chosen for reproduction, and their offspring are used to replace the individuals with the worst performance. A genetic-like crossover is used for recombination in which two parent chromosomes are are randomly 'spliced' and recombined to create an offspring chromosome. Over the course of many evolutionary steps, the thresholds generally converge on what is likely the maximum threshold achievable given the construction of the decoder and constraints of the code. The best single individual is selected and its chromosome is used to create the degree sequence polynomials for the top performing code ensemble.

The equations used for density evolution analysis contain many convolutions that need to be computed, as seen in (2.27). An efficient method for computing convolutions is to take Fourier transforms and multiply. This is how all convolutions are calculated in this work, and a few notes should be made about the effects of the Fourier Transform. Fourier transforms are efficiently computed using N-point DFTs where N is the level of quantization in the probability distribution, and the quantization error that results can affect the analysis. The value N must be large enough such that this error is small, and was chosen to be 2048 for these trials which was found to be sufficient through multiple attempts with different values for N. The C library FFTW was used to perform all Fourier transforms in this work [5].

With the density evolution tools in place, and a global search mechanism available to us, we are ready to optimize the different decoders, beginning with the sum-product decoder.

3.2 Optimization of the Sum-Product Algorithm

Using the GA setup discussed, optimally performing codes can be generated for the sum-product algorithm. Density evolution for this algorithm follows the procedure presented in [18] and the reader is guided to this paper for details. The performance of these codes is very good, and for larger block lengths approach their thresholds. This is the only decoding algorithm examined that consistently improves in performance as the maximal degree increases, so long as the number of iterations allowed is sufficiently large, and this behaviour is predicted by MacKay in [14]. On average, a large number of iterations is required to decode these highly optimized codes. The average number of iterations required rises to as high as several hundred for low SNR conditions. Again, the reader is directed to [18] for performance evaluation with unlimited iterations.

When a restriction on the maximum number of iterations is enforced, the performance of the sum-product decoder degrades, and Figure 3.1 provides more detail. These codes are derived from density evolution optimization with a constraint on the maximum number of iterations set to 200 and a block length of 10,000. For codes with unlimited number of iterations, performance of the Sum-Product algorithm improves as the maximal degree is increased. However, as can be seen from these results, this is not necessarily the case for constrained iterations.

There appears to be an error floor introduced by limiting the number of iterations which gets more severe as the maximal degree is increased, such that the performance of lower degree codes is better than that of larger degree codes for higher SNR conditions. For low SNR conditions, the higher degree codes outperform the inherently weaker lower degree codes, though still not nearly as well as comparable codes that are allowed a very large number of iterations. As the maximal degree is increased, variables on average participate in a greater number of parity check constraints, meaning that the decoding procedure requires more iterations for all of the received information to have a significant effect. As the number of iterations are allowed to grow for these codes, the error floor disappears and the standard 'waterfall' curve is seen as in Figure 3.2. The performance of these codes when allowed an unlimited number of iterations however, is worse than the codes that were optimized to

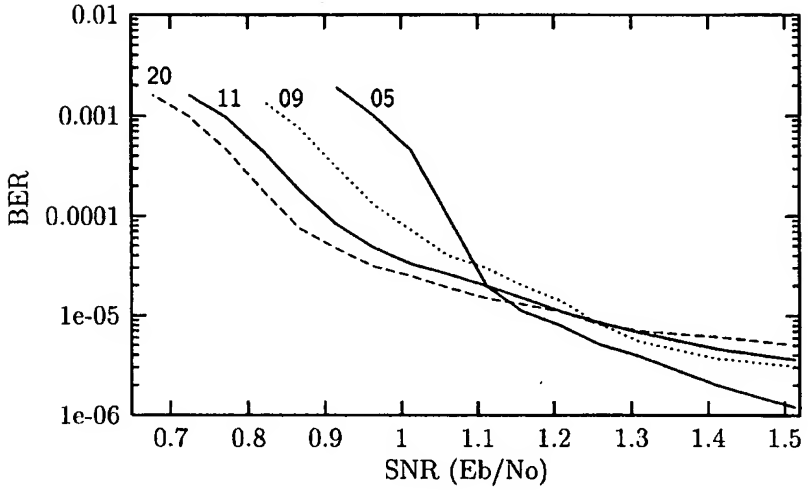

Figure 3.1: Sum-Product algorithm performance for rate 1/2 optimized codes of block length 10,000, maximum number of iterations=200. The numbers 05-20 indicate the maximal variable degree for each code used.

operate with a very large number of iterations.

The error floor problem becomes worse as the code's block length is shortened. It may be useful to couple these codes with a second high-rate code to remove the error floor while still reducing the overall number of computations required for successful decoding due to a decreased overall number of iterations. Alternatively it would be nice to be able to identify those sections of the graph that are still in error and concentrate the decoding effort on those sub-graphs only.

When the inherent power of the sum-product is limited by practical constraints, there is a significant degradation in the codes' performance, though overall, the performance is still excellent compared with traditional codes. We shall see in the next section that the min-sum algorithm can also be optimized for practical situations, and the reduced complexity decoder can perform nearly as well as the powerful sum-product algorithm.

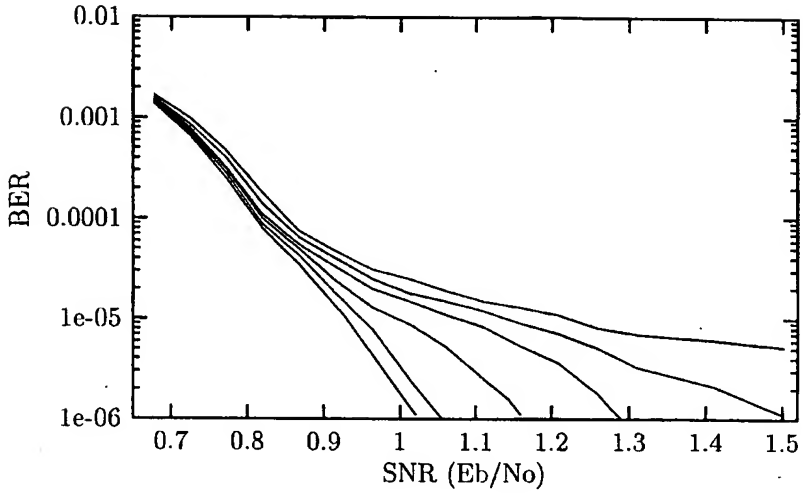

Figure 3.2: Effect of iterations on sum-product algorithm performance for optimized rate 1/2 codes of block length 10,000 and a maximal degree of 20. Iterations range from 200 to 1200 in increments of 200.

3.3 Optimizing the Min-Sum Algorithm

Most of the complexity in the sum-product algorithm arises from the need to calculate complex products for the check node update rule in (2.14). If this equation is closely examined, certain simplifications may be made to greatly reduce the complexity. Figure 3.3 (a) shows the input-output characteristics of the check node equation based on all inputs being equal, and (b) shows the situation for one input being much less than the rest. It is easy to see that for the case of one input being much less than the rest, output values are about equal to the input values, and even for all inputs being equal as in (a), the difference from the input=output line is not too great, and is bounded.

Using this knowledge, we can reduce computational complexity of the check node update rule through the use of an approximating function, and this will be called the *min-sum* algorithm. The check node function for the min-sum algorithm returns a value computed as follows: Let the magnitude of an output message be the minimum magnitude of all inputs excluding the edge whose output is being calculated. Let the sign of the output message be

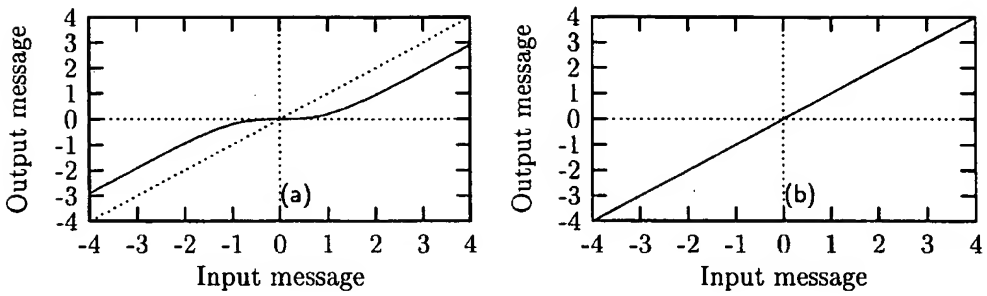

Figure 3.3: Check node output vs. input curves. (a) Input-Output characteristics of check node for equal inputs. (b) Input-Output characteristics of check node for one message much less than all others.

the modulo-2 addition of the signs of all the inputs excluding the edge whose output is being calculated. If the messages are represented by log likelihood ratios given in Table 2.1, this rule can be written as

$$\mu_{f\to x}(x) = \left(\prod_{y\in n(f)\setminus x} \operatorname{sgn}(\mu_{y\to f}(y))\right) \min_{y\in n(f)\setminus x} |\mu_{y\to f}(y)| \tag{3.2}$$

where f represents the check node in question, and x the variable which connects to f though the particular edge we are examining. The update rule for the variable side remains the same as for the Sum-Product algorithm in (2.13).

This simplification is a useful step, but we would also like to determine optimal threshold performances for this algorithm using density evolution concepts. In order to do this, the distribution of messages must be determined at the output of the check nodes. We already know how the density evolves at the variable nodes, and with the described update rules for the min-sum algorithm, the output cumulative distribution of messages from check nodes can be written

$$F(y) = \frac{1}{2}\left[1 - F_X(|y|) + F_X(-|y|)\right]^N \left[1 + (1 + 2F_X(0))^N\right] \tag{3.3}$$

where $F(y)$ is the cumulative distribution function of the log-likelihood ratio probability messages of all of the edges being input to check nodes. This function enables us to model

the evolution of probabilities on the graph over time. We may now proceed in finding degree sequence polynomials which have optimized performance thresholds.

Families of optimized codes were generated using a genetic algorithm. Degree sequences were limited to maximal degrees of 4, 5, 7, 9, 11, 13, 15, 20, and 25. The maximum number of iterations allowed was set to 25, 50, 100, and 200. For each case, an optimized degree sequence was returned by the GA. Table 3.1 shows the resulting degree sequences and thresholds for the cases in which iterations were limited to 100. Similar results were found when iterations were allowed to be greater or fewer, and in general, the thresholds found grow larger as the number of iterations are allowed to increase. Note that the thresholds given in Table 3.1 grow smaller as the maximal degree increases, and is seen with the sum-product algorithm as well.

The min-sum algorithm, as expected, does not perform as well as the sum-product algorithm. The thresholds found using density evolution optimization are greater than the sum-product algorithm, though the trend of decreasing thresholds with increasing maximal degree is still seen. Table 3.2 gives a comparison between the thresholds found for each algorithm when iterations are unlimited and limited to 100 iterations. Clearly, from a threshold perspective, the sum-product algorithm is superior; however it is useful to compare their performance in practical situations.

First, an examination of the performance curves for the Min-Sum algorithm reveals that for practical cases, larger maximal degree codes perform *worse* than smaller maximal degree codes for all SNRs. As an extreme case, one can see in Figure 3.4 that the performance for the code with a maximal degree of 25 performs much worse at all SNRs investigated than the code of maximal degree of 11, which was found to be the best performing code of the group. Since the min-sum algorithm does not compute exact APP values, errors which occur from the approximation are propagated through the graph. Check nodes with high degrees have a much greater chance of sending a larger error to many more nodes in the graph, and due to this inherent problem, performance of these codes suffer.

There is an error floor that can be seen with these codes and, like decoding with the sum product algorithm, is a result of the iteration constraint put on the codes. Coupled with the errors introduced by the approximation of the APP values, this results in an error floor

d_v	4	5	7	9	11	13	15	20	25
λ_2	0.369966	0.355097	0.316601	0.334492	0.302308	0.309376	0.297488	0.284456	0.267085
λ_3	0.411707	0.340715	0.400099	0.375263	0.321379	0.344362	0.265682	0.291418	0.312483
λ_4	0.218327	0.163818	0.076003	0.071285	0.034657	0.065113	0.089733	0	0
λ_5	0	0.140370	0	0.024016	0.026794	0.017904	0.089733	0	0
λ_6	0	0	0	0	0	0	0	0.051250	0.051886
λ_7	0	0	0.207297	0	0	0	0.065060	0.110688	0.046961
λ_8	0	0	0	0	0	0	0	0.041201	0.040485
λ_9	0	0	0	0.194945	0	0.146449	0	0.051250	0.023379
λ_{10}	0	0	0	0	0.174584	0.051684	0	0.035519	0.026411
λ_{11}	0	0	0	0	0.140277	0	0	0	0
λ_{13}	0	0	0	0	0	0.065113	0	0	0
λ_{14}	0	0	0	0	0	0	0.117540	0	0
λ_{15}	0	0	0	0	0	0	0.074765	0	0
λ_{18}	0	0	0	0	0	0	0	0.023530	0
λ_{20}	0	0	0	0	0	0	0	0.110688	0
λ_{23}	0	0	0	0	0	0	0	0	0.126607
λ_{25}	0	0	0	0	0	0	0	0	0.104703
ρ_5	0.652006	0.510473	0.108624	0.330566	0.028971	0.157893	0	0.182163	0.050973
ρ_6	0.347994	0.337985	0.885216	0.346133	0.057793	0.348033	0.415099	0.295964	0.176702
ρ_7	0	0.151542	0.006161	0.229296	0.083734	0.380856	0.441478	0.045857	0.183145
ρ_8	0	0	0	0.094006	0.829502	0.095024	0.143423	0.033483	0.381413
ρ_9	0	0	0	0	0	0.018193	0	0.155725	0.203786
ρ_{10}	0	0	0	0	0	0	0	0.286808	0.003980
$\left(\frac{E_b}{N_0}\right)^*$	1.221	1.213	1.157	1.089	1.040	1.016	0.998	0.967	0.914
σ^*	0.8688	0.8696	0.8752	0.8821	0.8871	0.8896	0.8914	0.8946	0.9001

Table 3.1: Good rate 1/2 degree sequences for the min-sum algorithm over the AWGN, optimized for 100 iterations. The values σ^* and $\left(\frac{E_b}{N_0}\right)^*$ are the threshold noise power and SNR for the given degree sequence.

Iterations	Maximal Degree	Sum-Product SNR^*	Min-Sum SNR^*
100	4	0.926	1.221
100	5	0.916	1.213
100	7	0.865	1.157
100	9	0.812	1.089
100	15	0.746	0.998
∞	4	0.782	0.967
∞	5	0.723	0.931
∞	7	0.502	0.787
∞	9	0.409	0.692
∞	15	0.335	0.651

Table 3.2: Comparison between sum-product and min-sum algorithm thresholds

that is severe for few iterations. Like the sum-product algorithm, the error floor becomes less severe, and disappears as the number of iterations increases.

The performance of the min-sum codes with respect to their threshold value also degrades faster for smaller block length than compared with the sum-product results. The reason for the degradation in performance is the effect of finite block lengths. Recall that threshold performance is the expected asymptotic performance as the codes length approaches infinity. The min-sum product algorithm is actually an approximation of the sum-product algorithm, and as such, these finite effects are much more pronounced, resulting in a rapid degradation in performance as block lengths decrease.

From the above discussion, we see that though the complexity of the min-sum algorithm is lower than the sum-product, the performance suffers significantly, and it is up to the code designer to decide which may be of more value. It is interesting to ask whether there is a better approximation function than (3.2) that is still computationally efficient. We will attempt to answer this in the next section.

3.4 The Modified Min-Sum Algorithm

The simplification of the check node update rule for the min-sum algorithm introduces an error into the output messages, which certainly impacts the performance of the decoder. This message error, $\Delta\mu$, defined as the difference between the actual output from a factor

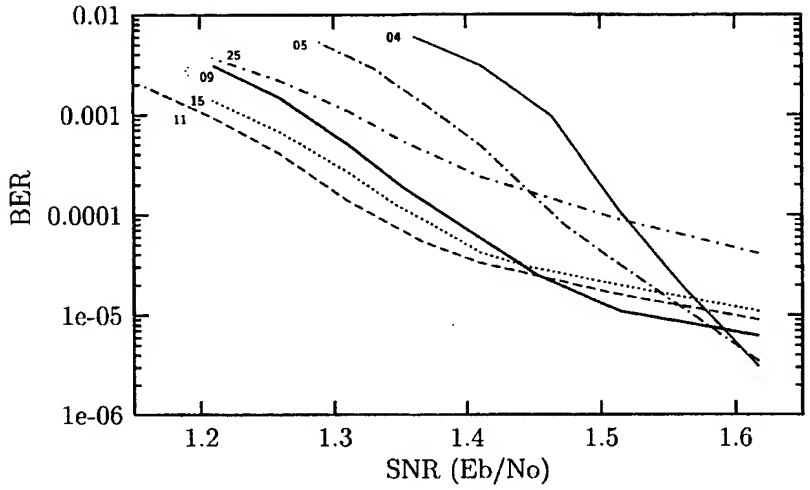

Figure 3.4: Optimized min-sum algorithm performance for rate 1/2 codes of length 10,000 with 200 maximum iterations.

graph update rule, μ_{actual} and the output from the min-sum update rule, $\mu_{min-sum}$, is

$$\Delta\mu \equiv \mu_{min-sum} - \mu_{actual} \tag{3.4}$$

and is bounded by

$$0 \leq \Delta\mu \leq \log d_c \tag{3.5}$$

where d_c is the degree of the particular check node. It is easy to check that no error will occur when all message magnitudes but the minimum are infinite, and that the maximal error occurs for the case where all input messages are of the same magnitude.

Knowing this bound, we can attempt to improve the min-sum decoding performance by introducing an additive term to the check node update rule. This approach has been used for other codes, and is very similar to the Max-Log-Map algorithm [17, 1, 20]. The goal is to closely approximate the transfer function of the check node as simply as possible, and one effective approach is to use a piece-wise linear approximation. A number of different limits and variations were tried for the approximation, and the best found is given here. Let $\mu_{initial}$ be equal to the result in (3.2), now let the resulting message output from a check node in

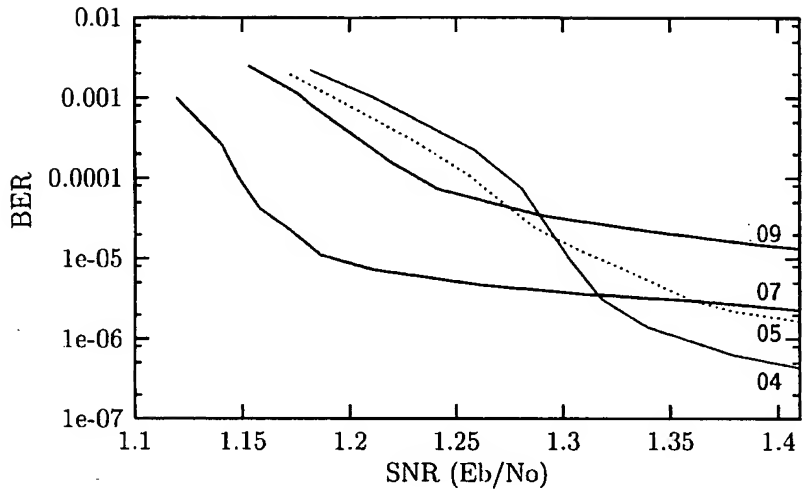

Figure 3.5: Performance of the modified min-sum decoder for rate 1/2 codes with block lengths of 10,000 and iterations limited to 200 for varying maximal degrees.

the modified decoder be

$$\mu_{f\to x} = \begin{cases} \mu_{initial} - K & \text{if } \mu_{initial} > K \\ \mu_{initial} + K & \text{if } \mu_{initial} < -K \\ \mu_{initial} & \text{otherwise} \end{cases} \tag{3.6}$$

where K is a constant value that is some fraction of $\log d_c$. The actual value of K used depends on the average d_c for that code, and was found to be effective when chosen to be $\frac{1}{3}\log d_c$. There are many alterations that can be made to this function by altering K or the limits. One could do a least squared fit to the transfer function, and the resulting approximating functions use would then depend on complexity and performance considerations. For the remainder of this discussion, only (3.6) will be used.

Once again, the ideas of density evolution may be applied to this modified algorithm to determine optimized degree sequences. The addition of the constant K complicates the expression for the probability density and numerical methods were used to calculate the required densities. It should be noted that the effect of the additional constant is to 'shift' those portions of the probability distribution towards the y-axis.

d_v	4	5	7	9	13	15
λ_2	0.369966	0.368436	0.343657	0.299828	0.272285	0.278550
λ_3	0.411707	0.398493	0.296163	0.280035	0.280872	0.268489
λ_4	0.218327	0.125519	0.204697	0.072121	0.004804	0.059741
λ_5	0	0.107552	0	0.067980	0.076561	0.041022
λ_6	0	0	0	0	0	0
λ_7	0	0	0.155483	0	0	0.044403
λ_8	0	0	0	0	0	0
λ_9	0	0	0	0.280035	0.004916	0
λ_{10}	0	0	0	0	0.116519	0
λ_{11}	0	0	0	0	0	0
λ_{13}	0	0	0	0	0.244042	0
λ_{14}	0	0	0	0	0	0.058201
λ_{15}	0	0	0	0	0	0.249594
ρ_5	0.652006	0.613466	0.191750	0.367716	0.153080	0
ρ_6	0.347994	0.296402	0.762445	0.008095	0.199490	0.319192
ρ_7	0	0.090131	0.045806	0.004716	0.004329	0.068649
ρ_8	0	0	0	0.619473	0.193456	0.612159
ρ_9	0	0	0	0	0.449646	0
$\left(\frac{E_b}{N_0}\right)^*$	1.221	1.197	1.128	0.793	0.735	0.709
σ^*	0.8688	0.8712	0.8782	0.9127	0.9189	0.9216

Table 3.3: Good degree sequences for the modified min-sum algorithm with rate 1/2 codes over the AWGN, optimized for 100 iterations. The values σ^* and $\left(\frac{E_b}{N_0}\right)^*$ are the threshold noise power and SNR for the given degree sequence.

Performance curves for the modified min-sum algorithm are given in Figure 3.5 and Table 3.3 shows the optimized degree sequences for the case of iterations limited to 100. The curves look similar to the regular min-sum algorithm with improved overall performance for lower SNR scenarios. The selected update equation has the effect of introducing many very low-value messages, and these messages are essentially considered erasures by the decoder. For larger maximal degrees, these small valued messages will be propagated through certain sections of the graph, resulting in some sub-sections of the graph becoming stuck in a non-convergent, inconclusive state. This is seen by the very prominent error floor in these codes. As with the other algorithms examined, this error floor gets much worse for larger maximal degrees, but a greater number of iterations mitigates the effect. Very high maximal degree codes were not investigated as it was assumed that they would have poor performance, and it appears that this is the case as the maximal degree 9 code is already worse than the maximal degree 7 code for all SNRs investigated.

The modifications to the min-sum algorithm do give the decoder better performance, but the error floor becomes more pronounced. We see again that there are trade-offs involved with the selection of a decoding algorithm. Depending on the target BER that is to be achieved, the modified min-sum may or may not be a good choice. We will examine some other factors affecting iterative decoder performance in the next section.

3.4.1 Comments on Good Degree Sequences

There are similarities between all of the optimized degree sequences, and it is worthwhile to examine them. The most striking commonality between all the optimized sequences, which is true for all of the algorithms, is the maximization of degree-2 variables. In all cases, the proportion of degree-2 variables is very close to the limiting value such that every check node has two degree-2 variable nodes participating with it. This means that the number of degree-2 variables is less than, but close to the number of check nodes. If the number of degree-2 variables is greater than the number of checks, it would be impossible for a code to be constructed without a cycle of length 2. It seems then, that the small degree variable nodes, counter intuitively are an important factor in powerful codes, as there are generally

a great deal of degree-3 variable nodes in these codes as well.

Besides this concentration, it also appears that good performance is achieved by having a large proportion of edges associated with the highest variable degrees. Besides the high number of edges participating with degree 2 and 3 variable nodes, a majority of the remaining edges tend to participate with the highest degree variables, especially in the case of the sum-product algorithm. These edges, however, seem to be the cause of the finite effects that degrade the practical performance of these codes, and there are generally fewer edges participating with these high degree nodes when they are optimized with iteration constraints.

3.5 Finite Effects

Fundamentally, there are two factors preventing the codes presented in the previous sections from achieving their threshold performances. The first is the finite block length of the codes, and the second is the finite number of maximum iterations allowed.

Both the min-sum and modified min-sum algorithms share certain commonalities which will be discussed. First, as with other iterative decoding schemes including turbo-codes, performance is highly dependent on the block length of the code. Figure 3.6 shows this relationship for the min-sum algorithm given a maximum of 200 iterations, and Figure 3.7 gives the same relationship for the modified min-sum algorithm.

In addition to the block length, both the Min-Sum and modified Min-Sum algorithms perform much better when given a greater number of iterations to decode. Figure 3.8 gives results for the Min-Sum algorithm, and similar results hold for the other decoding methods examined.

Finally, it is useful to investigate the nature of the error floor found in the various decoders. Figure 3.9 shows the error floor drop off for the three decoding schemes discussed at higher SNRs. An examination of Figure 3.9 shows that all three algorithms operate in three 'modes'. The first, at lowest SNR is the standard performance seen for these curves and performance is limited by the noise conditions. As the SNR increases, these codes enter an 'iteration limited' region of performance in which performance improvement with increasing

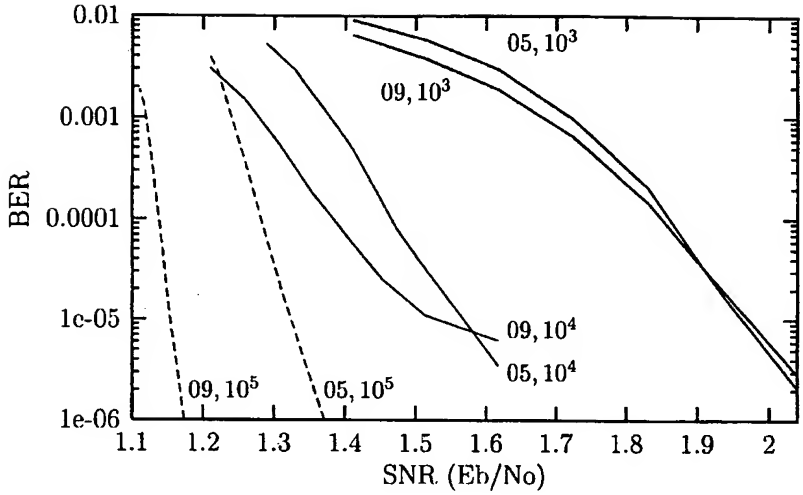

Figure 3.6: Performance of the min-sum decoding scheme for varying block lengths. All algorithms are highly dependent on the block length used. The first number is the maximal variable degree and the second is the block length. Iterations are limited to 200, and the rate is 1/2.

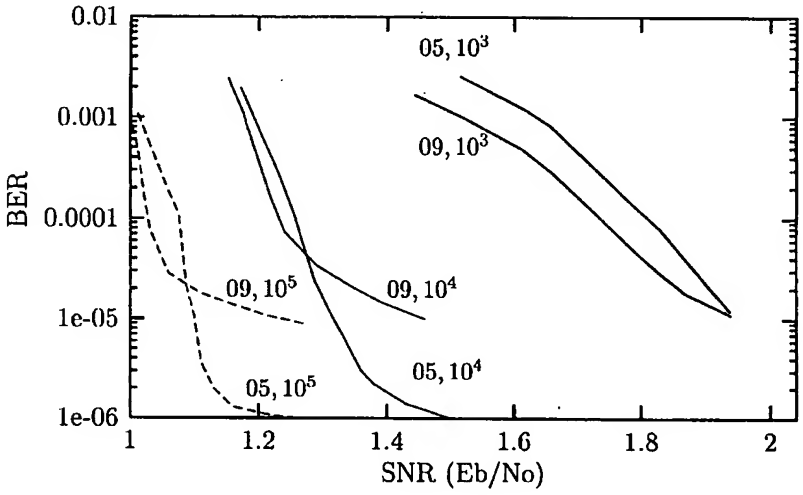

Figure 3.7: Performance of the modified min-sum decoder for maximum iterations of 200 for rate 1/2 codes of varying maximal degrees and block lengths.

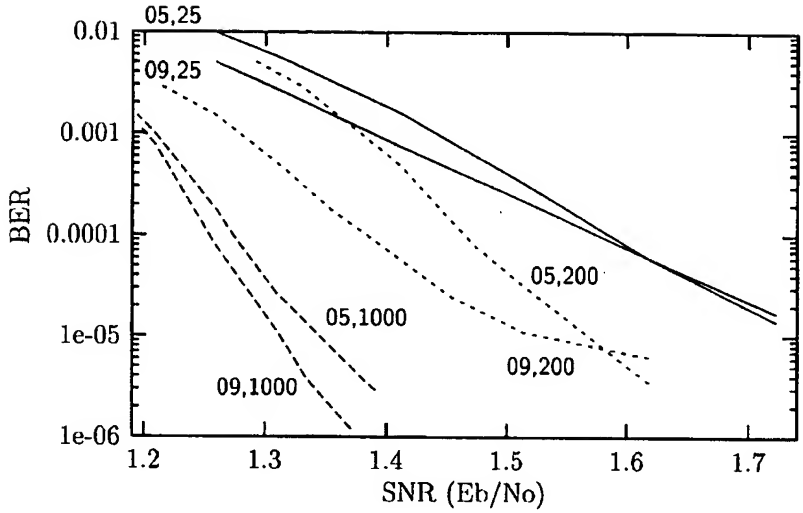

Figure 3.8: Performance of the min-sum decoder for block length of 10,000, for varying maximum number of iterations.

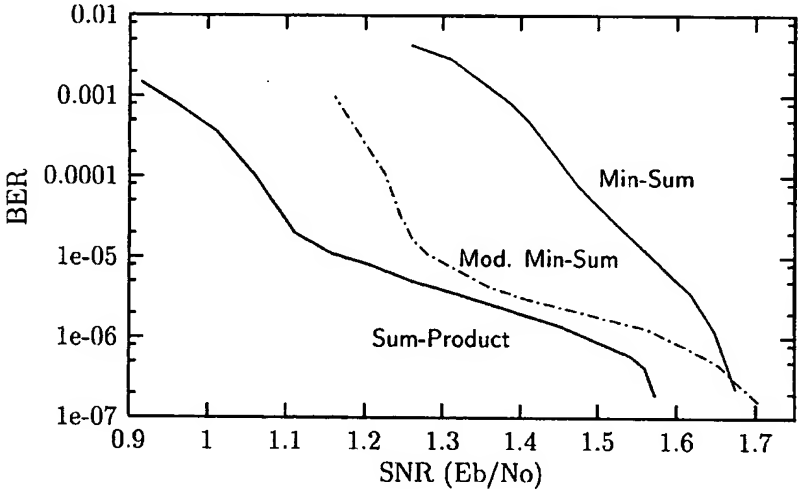

Figure 3.9: Performance of the three decoding schemes at high SNR for rate 1/2 codes showing the drop off of the error floor.

SNR is smaller than expected. Performance is limited by the fixed number of iterations allowed. Finally, at high SNR regions, performance improvement begins to increase as expected.

3.6 Performance vs. Complexity

We have discussed in previous sections some of the impacts that practical considerations, such as the maximum number of iterations allowed and block length, have on performance. It would be useful to quantify these trade-offs in a meaningful way to try to get a clear picture of the issues, and we accomplish this by plotting the performance of these decoding algorithms against the decoding cost in terms of their computational complexity.

A codes *true* complexity is not simple to calculate and is highly dependent on how it is implemented. The complexity calculated and used in these trials is based on the direct computational effort required only within the decoding algorithm, and this is the computations required at each node. All basic computations are considered to be of equal effort, so an addition is the same complexity as a compare or a multiplication. Also, overhead computational requirements such as loading and unloading of buffers is not included. The intention is to provide a rough measure that serves as a relative comparison rather than an absolute description.

Based on efficient calculations (i.e., using the FBA), we can derive expressions for the complexity of a given decoding procedure. At a node of degree D, the number of basic computations required for the FBA is $3(D-2)$. For the sum-product algorithm, 5 operations are required per basic FBA operation at a check node, such that the the complexity per bit of the algorithm, C, can be written as

$$C_{S-P} = I\,(\text{variable operations} + \text{check operations}) \tag{3.7}$$

$$= I\left(3 - \frac{5}{\int_0^1 \lambda(x)dx} + 5\left[3 - \frac{6}{\int_0^1 \rho(x)dx}\right]\right) \tag{3.8}$$

which simplifies to

$$C_{S-P} = I\left(18 - \frac{5}{\int_0^1 \lambda(x)dx} - \frac{30}{\int_0^1 \rho(x)dx}\right) \tag{3.9}$$

where I is the number of iterations used and was set to be one of 25, 50, 100 or 200, depending on how many maximum number of iterations are allowed. The min-sum algorithm requires an additional select operation at each check node beyond the basic computations, so its complexity per bit can be written

$$C_{M-S} = I\left(6 - \frac{5}{\int_0^1 \lambda(x)dx} - \frac{4}{\int_0^1 \rho(x)dx}\right) \tag{3.10}$$

and similarly for the modified min-sum which requires an additional compare and add per bit at each check node,

$$C_{M.M-S} = I\left(8 - \frac{5}{\int_0^1 \lambda(x)dx} - \frac{4}{\int_0^1 \rho(x)dx}\right). \tag{3.11}$$

Figures 3.10 and 3.11 give the performance complexity trade-off for BER targets of 10^{-3} and 10^{-5} respectively. The curves represent codes of block length 10,000 rate 1/2, and maximum iterations limited to 200. For all of the cases examined, there is an optimal operating point, meaning that for some added complexity in the form of higher maximal degrees, decoding performance actually degrades. This is an important result for practical situations. What seems to be happening in general is that the highest degree nodes require a very large number of iterations to be decoded correctly, and for situations where there is a practical limit on these, the code actually performs worse.

For a target BER of 10^{-3}, one can see that there is no obvious 'win' in terms of the performance complexity trade-off between the three decoding schemes, although it seem one can achieve higher performance with lower complexity for some cases with the modified min-sum over the standard min-sum algorithm. For a target BER of 10^{-5}, we see a clear win with the modified min-sum algorithm over the standard min-sum, and it even outperforms the sum-product algorithm in certain instances.

Figure 3.12 shows the performance complexity trade-off for a target BER of 10^{-5} with iterations constrained to 25. The performance is worse than for the case of maximum iterations set to 200, but the general pattern remains. Note that the lower maximal degree codes perform relatively better with fewer iterations. For example, the best performing code for the modified min-sum algorithm with 25 iterations has a maximal degree of 5, whereas it is 9 for 200 iterations.

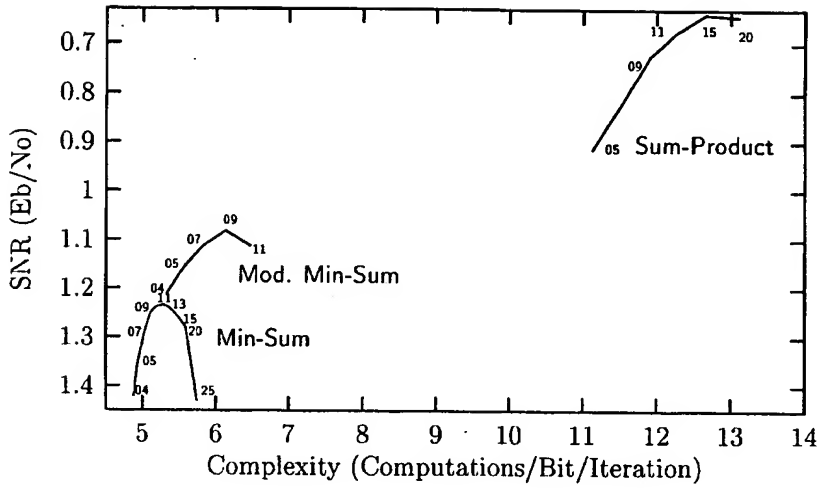

Figure 3.10: Performance Complexity trade-off for a target BER of 10^{-3} for the sum-product, min-sum and modified min-sum algorithms.

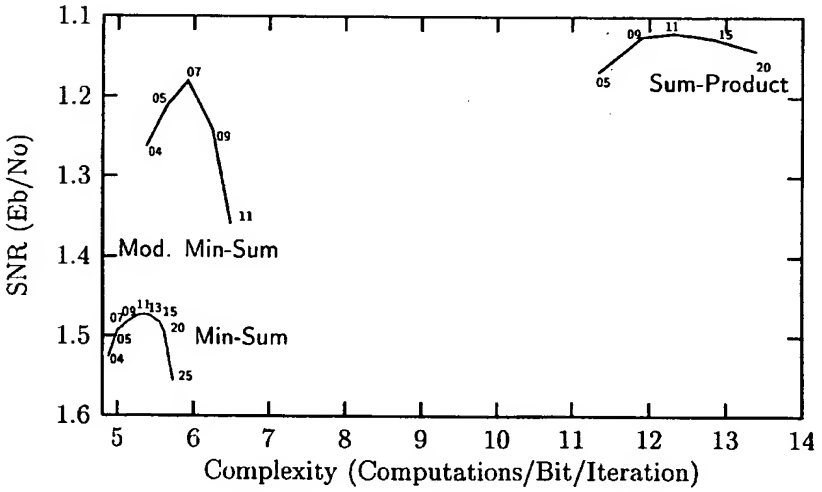

Figure 3.11: Performance Complexity trade-off for a target BER of 10^{-5} for the sum-product, min-sum and modified min-sum algorithms.

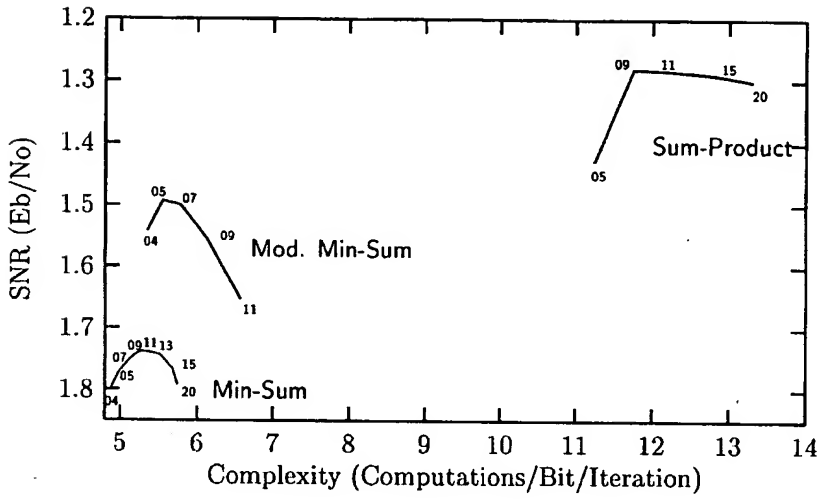

Figure 3.12: Performance Complexity trade-off for a target BER of 10^{-5} for the sum-product, min-sum and modified min-sum algorithms with maximum iterations set to 25.

It is useful to understand the relationship with varying block lengths for these algorithms, so two plots are given in Figures 3.13 and 3.14 for comparison, having block lengths of 100,000 and 1000 respectively. As the block length increases, the codes become more sensitive to the maximal degree parameter. The difference between the performance of the optimal code versus one with a smaller or larger maximal degree is much greater for block lengths of 100,000 than 1000, especially for the min-sum and modified min-sum decoders.

We have seen that for practical cases, reduced complexity decoders such as the min-sum and modified min-sum provide positive alternatives to the sum-product algorithm in terms of decoding power per unit operation. Overall performance does suffer however, and we would like to see ultimately a practical, realizable decoder that can perform as well as the sum-product without the complexity. The answer, perhaps, may be found by examining the actual implementation of the decoder and exploiting the power of the platform.

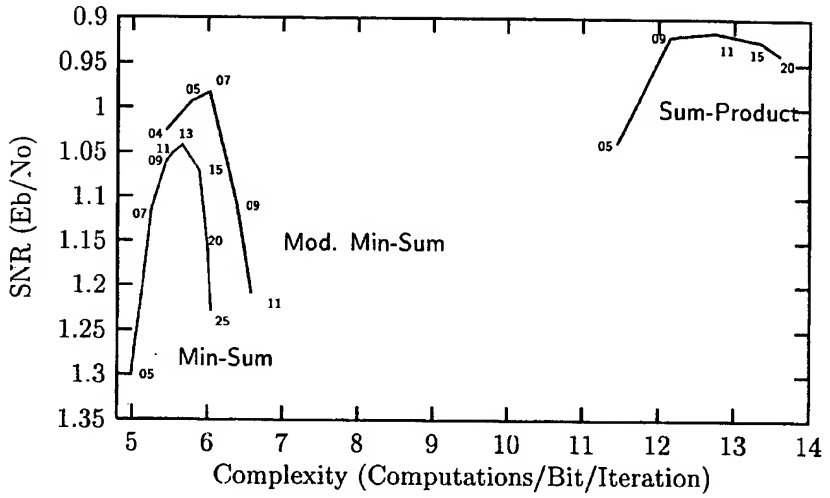

Figure 3.13: Performance Complexity trade-off for a target BER of 10^{-5} for the sum-product, min-sum and modified min-sum algorithms for block length of 100,000.

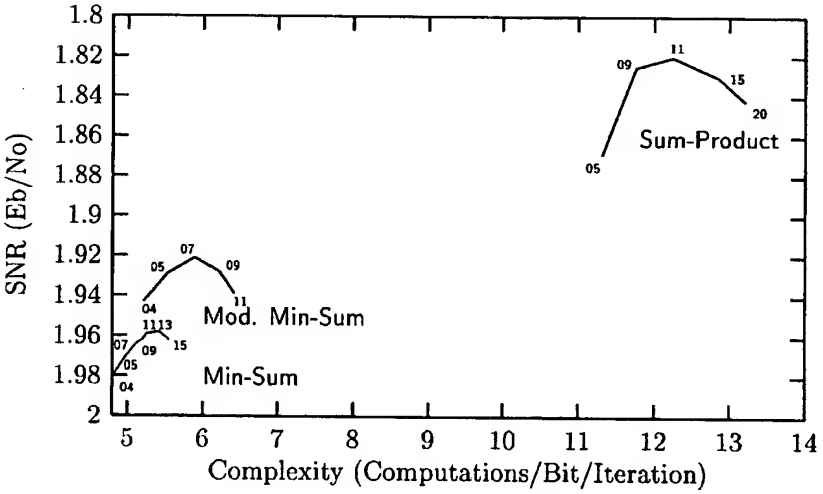

Figure 3.14: Performance Complexity trade-off for a target BER of 10^{-5} for the sum-product, min-sum and modified min-sum algorithms for block length of 1000.

Chapter 4

The Sum-Transform-Sum Algorithm

Applications that require very powerful FEC schemes can greatly benefit from LDPC codes, as the performance is beyond most other coding choices. One difficulty with the LDPC codes presented so far is that they are not necessarily well suited for implementation in devices such as modems. These codes have been developed with decoding performance in mind, with no thought to the application.

A different approach can be used to find a good decoding algorithm. First, understand the implementation, and use the features inherent in the implementation to find powerful decoding algorithms. Applied to a hardware instantiation, this approach leads to the *sum-transform-sum* (SXS) algorithm.

4.1 Algorithm Description

The SXS algorithm is a hardware architecture and algorithm for a soft-input, soft-output iterative decoder for LDPC codes, and is similar to the standard sum-product or min-sum algorithms. The inputs to the algorithm are soft representations of the probabilities for the possible transmitted symbols, and for a hardware implementation, it is natural to represent these values as fixed-point numbers. The precision with which these number will be represented is a parameter of the system and will depend on, among other things, memory constraints, and hardware complexity.

Unlike the other iterative decoding schemes described, the SXS algorithm utilizes two separate probability representations for messages and requires a transformation algorithm to convert between the two. The two representations for messages used are the LLR and the LLDP, both of which have been previously described and defined. Variable nodes work in the LLR domain while check node operate in the LLDP domain. Using these representations, both the variable node update equation and check node update equation are simple to calculate, being the summations of (2.13) and (2.15) respectively.

It is useful to note that these two representations are *duals*, meaning that decoding in one domain is equivalent to decoding the dual code in the other domain. The idea of using the dual domain to decode linear codes was proposed originally by Hartmann and Rudolph [7]. Forney has suggested the use of dual codes and a Fourier Transform method to convert between them [9]. The use of the dual code can reduce the complexity involved in the decoding procedure without any performance loss, however the conversion between domains may require intensive computations.

With two separate probability representations, the SXS algorithm requires a function $\Gamma(x)(s,r)$ to transform messages between them. Let μ_{ratio} be a message in the log-likelihood ratio domain, and μ_{diff} be a message in the log-likelihood difference pair domain. The function $\Gamma(x)(s,r)$ takes as its argument a message of the form μ_{ratio} and returns a message of the form μ_{diff}. The returned message has the equivalent probability as the input message, only expressed in a different manner. Similarly, the inverse transform function $\Gamma^{-1}(s,r)(x)$ takes as its input a message of the form μ_{diff} and returns the equivalent probability in a message of the form μ_{ratio}. The function $\Gamma(x)(s,r)$ may be written as

$$\Gamma(x)(s,r) = \left(\operatorname{sgn}\left[2\left(\frac{e^x}{1+e^x}\right)-1\right], \left|\log\left|2\left(\frac{e^x}{e^x+1}\right)-1\right|\right|\right) \tag{4.1}$$

where x is a message of the form μ_{ratio}. Likewise, the inverse transform function $\Gamma^{-1}(s,r)(x)$ may be written as

$$\Gamma^{-1}(s,r)(x) = \log\left(\frac{1+se^r}{1-se^r}\right) \tag{4.2}$$

where (s,r) is the sign and magnitude of a message of the form μ_{diff}.

The function in (4.1) and its inverse, (4.2), may be implemented very simply using a pair of look-up tables in memory. If, for example, messages are stored as 8-bit (N-bit in general)

values, then two look-up tables each of size $2^8 = 256$ bytes (2^N N-bit values in general) are required to completely describe the forward and inverse functions. One need only to calculate the value of the transform for each possible input once, and store the results in memory for future use. Alternatively, the function and its inverse may be used themselves to calculate the necessary transforms on an as-needed basis.

It may also be possible to implement the transformation through the use of a combination of logic and memory. For example, it may be a simple matter to use some logic to determine the most significant bits of the transform output, and determine the least significant bits in a reduced look-up table. This flexibility allows the algorithm to be used in an optimal manner depending on the platform that is it is implemented on. If processing speed is a constraint, it would make sense to take advantage of the look-up table as much as possible as opposed to calculating multiple transforms.

One difficulty in using fixed-point precision for this algorithm is the many-to-one, one-to-many problem, which occurs as a result of the non-linear nature of the transformation in a fixed-point system. Multiple values in one representation may map to the same value in the other and vice versa, making the choice of ranges and precision for these values critical. For example, the values 7, 8, and 9 may all map to the value 1 in the other domain, and the problem is that information is lost in mapping back from 1 to the original domain.

Another issue with fixed-point implementations is the range of values that may be taken on by the precision used for the system. If too few bits are used to represent the values, performance may be very poor. In fact, this is exactly the case. When 6 bits are used to represent the data, the decoder works very poorly indeed, but as the precision is increased to beyond 12 bits in general, performance is limited by the power of the code itself, not the fixed-point precision.

The two message representations, μ_{ratio} and μ_{diff}, have different domains, as shown in Table 2.1, but since a sign bit is naturally represented in the fixed-point world of hardware, the domains become the same, and this is finitely limited by the choice of how decimal bits and whole bits are allocated within the overall bits available. For effective performance, μ_{ratio} messages may need to take on very large values, whereas μ_{diff} messages may have very small decimal values. It is advantageous to allocate more whole bits to μ_{ratio} messages, and

Precision	μ_{ratio} Allocation	μ_{diff} Allocation
6-bit	1-3-2	1-1-4
8-bit	1-3-4	1-1-6
10-bit	1-4-5	1-1-8
12-bit	1-5-6	1-2-9
16-bit	1-6-9	1-3-12

Figure 4.1: Bit allocation for messages in the SXS algorithm. The first digit is the number of sign bits, then the number of whole bits, then the number of decimal bits

more decimal bits to μ_{diff} messages. Figure 4.1 shows how bits are allocated for varying precision for the trials performed with these codes.

4.1.1 Decoding

The decoding algorithm commences by updating the check node output messages. This is accomplished using the following equation:

$$\mu^i_{ratio} = \Gamma^{-1}\left(\prod_{x\in n(i)} s_{\mu^x_{diff}}, \sum_{x\in n(i)} r_{\mu^x_{diff}}\right) \tag{4.3}$$

where i is the ith edge on the graph and $n(i)$ is the set of edges connecting to the check node associated with i, *excluding* i itself. The variables s and r represent the sign and magnitude for the pair (s,r) which is how μ_{diff} messages are represented. This equation is used a number of times based on the structure of the LDPC code to update all of the elements of the variable input array. In order to increase the efficiency of this equation, the FBA may be used [11], or an efficient one-pass algorithm may also be used [8].

Once completed, the variable node edges are then updated. The rule for updating these messages is governed by the equation

$$\mu^j_{diff} = \Gamma\left(\mu^j_{init} + \sum_{x\in n(j)} \mu^x_{ratio}\right) \tag{4.4}$$

Code	Iterations	Threshold
Regular $(3, 6)$	∞	1.13dB
Optimized Max Degree 4	200	1.04dB
Optimized Max Degree 4	50	1.26dB
Optimized Max Degree 5	200	0.97dB
Optimized Max Degree 5	50	1.16dB

Table 4.1: Threshold performance for SXS codes

where j is the jth edge in the graph and $n(j)$ is the set of edges connecting to the variable node associated with j, *excluding* j itself. The order in which the variable and check node input arrays are updated may vary depending on the actual code used. These messages are iteratively updated until some convergence criteria is met, as with the other algorithms explored in this paper.

4.2 Optimization and Performance

Density evolution can be used to find optimized codes for this algorithm, and the equations involved are actually identical to those used for the sum-product algorithm. The sum-product algorithm, however, uses floating-point values, whereas the SXS algorithm operates in the fixed point domain. As such, density evolution needs to also operate in fixed point. Fixed-point number precision adds another dimension into the search space for good codes. Trials were performed at 10-bit and 12-bit precision and all results presented here are from the 12-bit case.

Like the cases for the other decoders optimized in this paper, the SXS algorithm was optimized for block lengths of 1000, 10000, maximal degrees of 4 and 5, and iterations of 50 and 200. For a comparison reference, a regular $\{n, 3, 6\}$ LDPC code is also presented. The threshold for the regular code under sum-product decoding is 1.13 dB. Thresholds for the optimized codes are presented in Table 4.1.

The performance for a regular $\{10000, 3, 6\}$ code is presented in figure 4.2. Given that

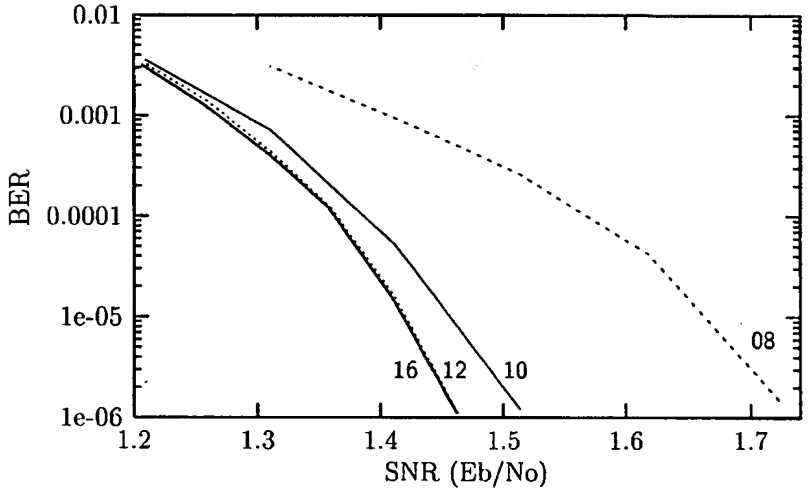

Figure 4.2: Performance of the SXS algorithm for a regular $\{10000, 3, 6\}$ code with maximum iterations set to 200. The effect of varying precision is also shown.

the theoretical threshold for this regular code is 1.13dB, the performance of the decoder is very good, especially considering the limitation on the number of iterations. Note that the performance of the decoder degrades rapidly for fixed-point precisions of less than 10-bits. The performance for 8-bit precision is poor, and the 6-bit performance would not even be visible on this graph. Once a limit is reached, at about 12 bits, there appears to be little additional gain by greater precision.

Figure 4.3 shows the performance of this decoder for a regular $\{1000, 3, 6\}$ code. The performance is worse than the longer code presented in figure 4.2, as is expected. For comparison with the sum-product algorithm, figure 4.4 shows the performance of both the SXS decoder performance and sum-product decoder performance for the same regular code. We see from the plot that the sum-product algorithm performs better, though not by much.

Genetic algorithms were used to find good degree sequences for the case of 200 maximum iterations, with maximal variable degrees of 4 and 5, these two being chosen to keep overall complexity to a minimum. The performance for these two codes and the optimized sum-

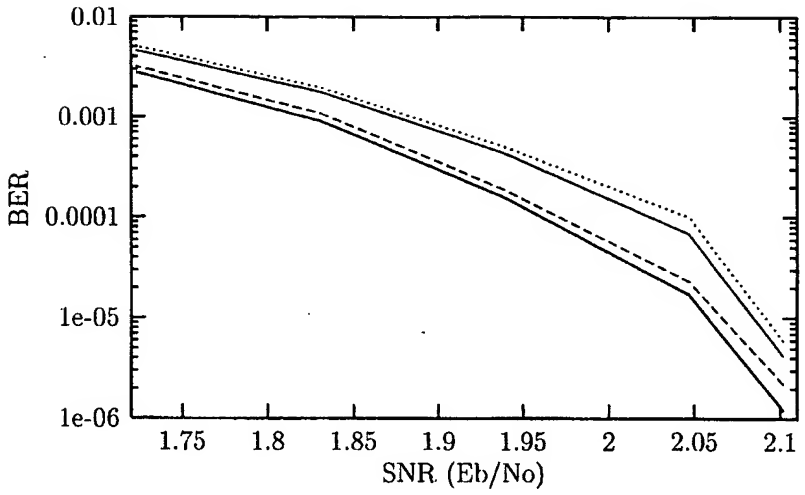

Figure 4.3: Performance of the SXS algorithm for a regular $\{1000, 3, 6\}$ code versus varying maximum iterations. The effect of varying precision is also shown. The curves, from worst to best, are: 10-bit: 50 iterations, 10-bit: 200 iterations, 12-bit: 50 iterations, 12-bit: 200 iterations

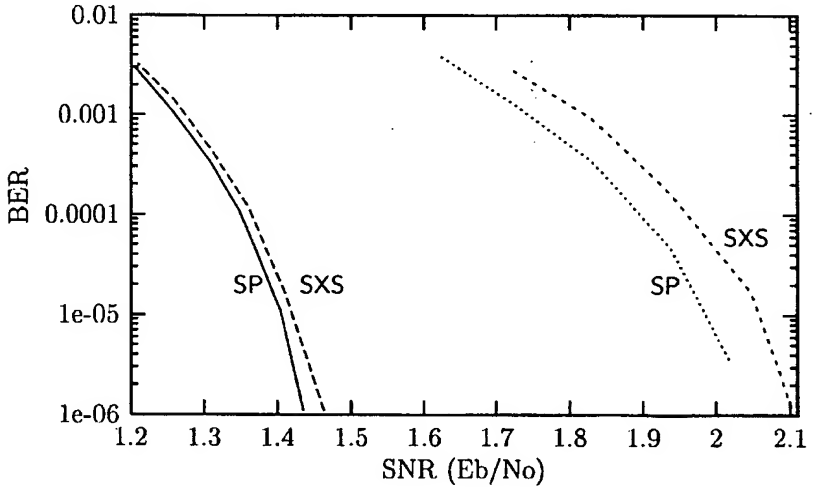

Figure 4.4: Comparison between SXS and sum-product performance for regular $(3, 6)$ codes. The two left curves are block lengths of 10,000, the two right curves are block lengths of 1000. In both cases the sum-product decoder is more powerful.

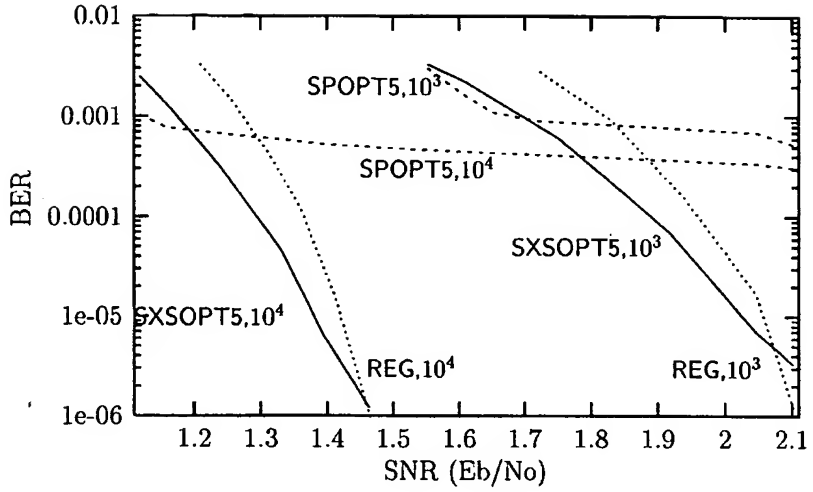

Figure 4.5: Performance of optimized SXS codes with maximum iterations set to 200 of lengths 1000 and 10,000. REG refers to the regular (3, 6) code, OPT5 and SXS-OPT5 refer to the optimized codes of maximum variable 5 for the sum product optimization and SXS optimization respectively.

product codes of Section 3.2 are seen in Figure 4.5. Surprisingly, the optimized sum-product codes perform very poorly when decoded using the SXS algorithm. Even in the higher SNR regions, most codewords did not decode correctly. Instead, two or three bits got 'stuck' in an incorrect state. The reason for this is the quantization of values in the SXS algorithm. The highly optimized sum product codes get stuck at fixed points since the incremental gains made in these regions are quantized away. The optimized SXS codes however, perform well, and their performance may possibly be made better given a more advanced transformation technique.

Solving the transformation mapping problem is not straightforward, and a few approaches were attempted besides simply increasing the precision of the fixed-point values. A 'pessimistic', 'realistic', and 'optimistic' mapping scheme were all used. The pessimistic approach assumed that in the case of a one-to-many mapping, that the transformed values magnitude was the smallest magnitude that would be mapped to that value. Similarly, the realistic scheme used an average value, while the optimistic scheme used the largest value. None of the attempts tried yielded a significant improvement in performance, and the realistic

scheme was selected for the trials.

This mapping problem is similar to the dynamic range problem associated with linear amplifiers, and a useful solution in that realm is companding. It may be possible to further mitigate the effects of this problem borrowing the companding technique. Companding is allocating the bits in a non-linear manner in such a way that small values are not affected too much by quantization error while large values do not extend beyond the range of the fixed-point number. This is an interesting area for future research for this algorithm.

The presented figures provide a good indication of the performance of this decoding algorithm. Given that the SXS algorithm was developed first with the ideas of implementation and simplicity, and then looking toward performance, it is a positive sign to see that the code performs well in general. We shall now examine the complexity of this algorithm.

4.3 Complexity

Having been designed with reduced complexity in mind, and having seen that the performance of these codes can be quite good, it is important to understand the overall complexity that is involved in the practical implementation of the SXS algorithm. To this end, we will calculate the computational power required per bit decoded.

At each computational node, we assume that the FBA is used. It is also assumed that the hardware that the algorithm is implemented on is able to handle the bit-depths required with one operation, e.g., 12-bit memory look-ups require one operations, as do 10-bit additions. Taking these assumptions into account, the number of computations required per bit is

$$C_{SXS} = I\left(8 - \frac{5}{\int_0^1 \lambda(x)dx} - \frac{6}{\int_0^1 \rho(x)dx}\right) \tag{4.5}$$

It is difficult to do a straight comparison with the decoders discussed in Chapter 3 due to the fixed vs. floating point computation differences. The computations required per bit for the SXS algorithm are presented in Table 4.2. If we assume a one-to-one complexity between fixed and floating point computations, we can compare the optimized SXS algorithm decoder with an equal complexity min-sum or modified min sum decoder. Plotting the performance-complexity point in Figure 4.6 we see that the overall performance is superior to the min-sum

Code	Complexity	SNR for 10^{-3} BER	SNR for 10^{-5} BER
Regular (3,6)	5.33	1.27dB	1.41dB
Optimized Max Degree 4	5.06	1.18dB	1.33dB
Optimized Max Degree 5	5.55	1.14dB	1.28dB

Table 4.2: Complexity and target SNR for SXS Algorithm codes of length 10,000. Complexity is presented as computations per bit per iteration.

decoder with similar complexity, and is not quite as good as the optimal modified min-sum decoder, although the decoding complexity is lower. This situation remains true at lower target BER as well. Given this pessimistic comparison between fixed-point and floating-point complexity, it is encouraging to see that as a worst case, the SXS algorithm performs quite well for its complexity. A more realistic comparison will only show an overall better outcome in favor of the SXS decoder.

4.4 Implementation Architecture

The SXS algorithm has been designed with the intention of being easily implemented in a fixed-point hardware environment, and we have seen that with this goal in mind good performance is still possible. The use of a non-optimized, regular code avoids the problem of error floors, and their constant degree in both check and variable nodes will simplify the structure of the hardware required to implement the algorithm. We now provide an example of a hardware implementation.

Let the code be a regular $\{1000, 3, 6\}$ code, therefore the number of edges on the graph is 3000. Figure 4.7 shows a block diagram for a possible implementation, and is the used for this example. Assume that there is a method to convert channel information into fixed-point probability messages (μ_{ratio} messages), and store the received word in an array called Input. There will be two sets of messages that will be needed, and the memory requirement for these are each of size 3000 and are the check and variable message memory blocks. There is also the need for two look-up tables to perform the necessary transformations from one message

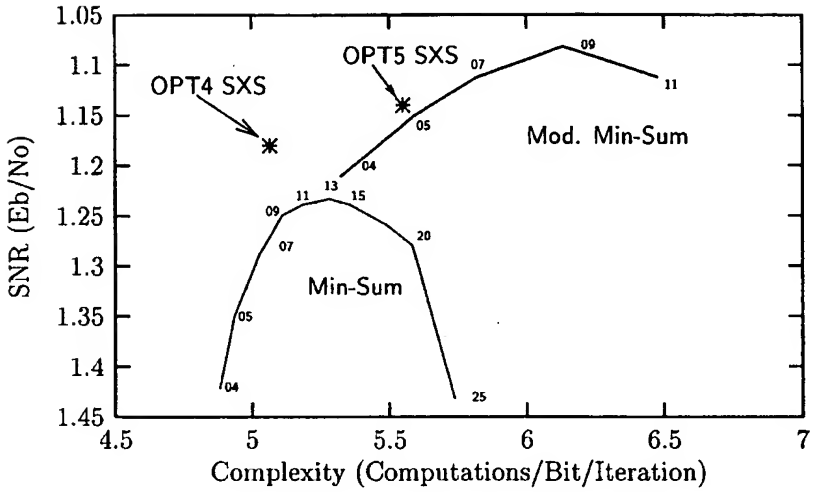

Figure 4.6: Performance-Complexity comparison of the SXS algorithm with floating-point algorithms. Target BER performance is 10^{-3}.

domain to the other, called the check and variable node transform blocks. The memory requirements for these arrays is entirely dependent on the fixed-point precision used, and if we assume 8-bit precision, we require a memory of size $2 \times 2^8 = 512$ bytes.

The decoder must know which edges participate at each of the nodes, and this is achieved through the use of the check and variable node descriptors, both of length 3000. Each of these arrays contains a list of how each edge is connected to its neighbour, and provides the information required to perform the update rules. Note that there is only a need for one of these arrays, as the information from one can be determined from the other, but the format in which the information is stored in both allows for a simple approach to updating both check and variable nodes.

Once the input array is initialized, the variable message memory block is set to be erasures, and the decoding process may commence. The variable node descriptor is stepped through and in groups of three, the FBA is used to produce three output messages which are transformed and written to the proper locations in the check message memory block. A similar process is used to then write new variable messages. The process repeats until some

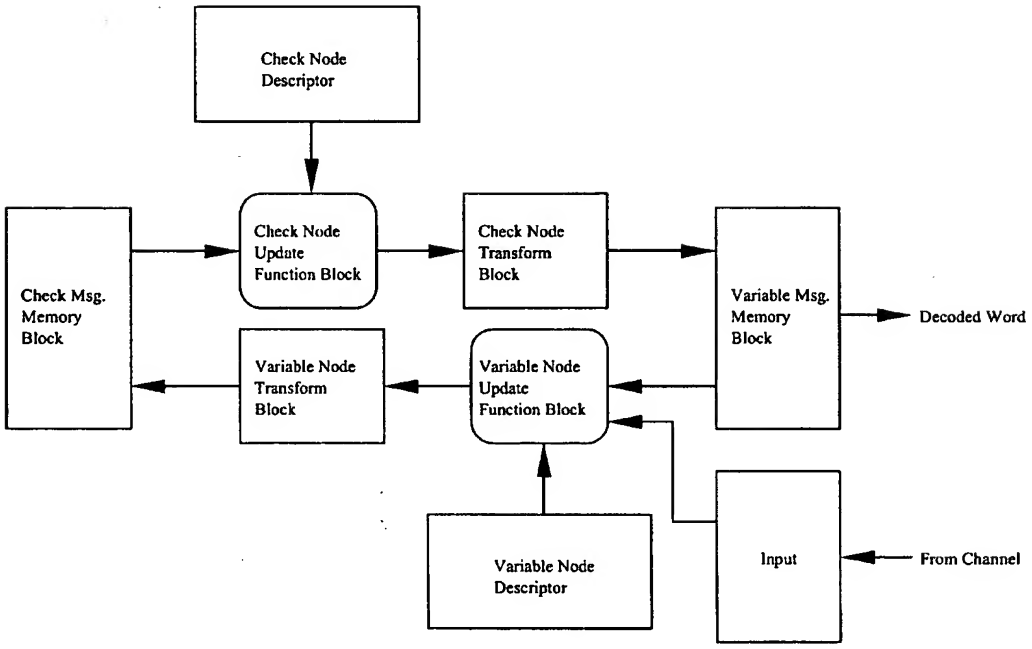

Figure 4.7: Block diagram of SXS algorithm architecture. Square blocks represent memory locations, rounded boxes are computation nodes.

termination criterion is satisfied, which is not shown in Figure 4.7. The criterion may simply be some counter tracking the iterations, or may be a test to determine if a valid codeword has been reached. After the decoder has terminated, the decoded word is output. This may be a soft decision or a hard decision, but some decision is made and is output. The word may be used from that point in future processing.

This relatively straightforward implementation requires an overall memory of size 13,512, or about 13k. In general, the amount of memory M needed for this implementation for a code of length n and a bit precision of b is

$$M = 4n \int_0^1 \lambda(x)dx + 2^{b+1} \tag{4.6}$$

It is also useful to look at the throughput capabilities for this architecture. The number of computations per bit per iterations is given in (4.5). Given a rate $R = \frac{1}{2}$ code, and limiting the number of iterations to 25, the number of computations required per information bit is

$$C_I = 25 \times 5.33 \div \frac{1}{2} = 266.5 \tag{4.7}$$

In a completely serial implementation such as the one presented, if the processor was capable of 40MIPS, then the throughput would be approximately 150kbps. As a very rough comparison, convolutional codes can achieve about the same throughput on a similar system, though with fewer memory requirements [21]. However, these codes do not necessarily perform as well.

One of the benefits of LDPC codes is their inherent parallel structure. This structure can be taken advantage of in a hardware implementation to greatly enhance the throughput of the system. It would not be difficult to introduce parallel update function blocks to speed up the decoder, and although doubling the number of function blocks wouldn't necessarily double the throughput, it would increase it dramatically as most of the computations required take place in these blocks.

Chapter 5

Conclusions and Future Work

Low Density Parity check codes were shown to be the most powerful codes found to date, approaching to within a fraction of a decibel of the Shannon limit. We investigated the properties of LDPC codes and examined the notion of density evolution. This powerful analytical technique allows the performance of LDPC codes to be predicted and optimized under certain assumptions, and empirical evidence has supported these claims.

Reduced complexity LDPC decoders were presented. Three decoding algorithms, the sum-product, min-sum, and modified min-sum algorithms are described, and a technique for optimizing their performance was developed. Density evolution was presented as a tool to maximize the performance of a given decoding scheme and was applied successfully to the three algorithms discussed. This approach may be used for any iterative decoding scheme that may be represented on a graph. We explored some of the issues affecting the performance of these codes including maximal degree, number of maximum iterations allowed, and block length of the codes. The sum-product algorithm was found to be the most powerful decoder, but also the most computationally intensive.

Performance-complexity issues were examined, and the trade-offs clearly presented. For all the decoders, there was a limit, at least in practical decoding scenarios, where increasing the computational effort no longer increased the performance of the code. This important result illustrates that an optimum operating point can be found for these decoders. While the sum-product algorithm provided the best performance, the modified min-sum algorithm

outperforms it in certain situations, and with many fewer computations required per bit. The min-sum algorithm, the least complex of the decoders presented performed nearly as well as the modified min-sum algorithm and may be an advantageous scheme for applications requiring low complexity.

The sum-transform-sum algorithm was described as an attempt to build a powerful yet simple LDPC decoder by exploiting the efficiencies found in the implementation platform, in this case, hardware. The fixed-point representations, and access to memory provided the means for the success of this approach. The performance of the SXS algorithm was good, but very dependent on the fixed-point precision and transformation look-up tables. This algorithm is a very viable solution for the implementation of an LDPC decoder in hardware, and a description of an actual implementation was detailed.

5.1 Suggestions for Future Research

LDPC research is at an exciting and vibrant stage; there are a great number of advancements being made, and thoughts are turning more towards the application of these powerful codes. The use of sparse matrix techniques has improved the encoding problem for these codes, but it would be nice to find a truly linear time encoding procedure. This problem may be solved by exploiting irregular codes and their tendency to have many low weight variables. This would allow the parity check matrix to be constructed in a manner to be encoded in linear time.

We have seen the effective use of practical decoding systems and have presented an algorithm capable of being easily implementable in hardware. Many applications could take advantage of this system, including the use of LDPC codes for wireless data transmission, satellite communications and storage media such as disk-drives. Third generation (3G) cellular systems have discovered the power of iterative decoding schemes and are writing turbo-codes into their specifications. LDPC codes could also be used with the same performance, and have the added advantage of being patent free, thus no royalties are required for their use.

As an extension to the work presented in this paper, it would be useful to examine a

few areas in more detail. There may be greater gains in performance made by tweaking the modified min-sum algorithm. If a simple yet accurate representation can be found for the check node update rule, the algorithm may be made more powerful. Perhaps it is possible approach the problem from a stochastic viewpoint and find a probabilistic approximation to the transfer function. It is also useful to investigate the error floor that results from optimizing the codes and finding a balance between performance and the severity of the error floor.

The effect of cycles in LDPC codes and efficient LDPC code construction techniques are also a useful area for further work. There is still no analysis available to determine the effect that cycles have on codes, though we know that it is detrimental to performance. There are likely certain construction techniques that may be used to mitigate the existence of cycles without affecting the probabilistic balance required for optimal decoding power.

Finally, further investigation into some of the negative aspects of the SXS algorithm would be instructive. The performance of these codes could possibly be much better if there were a practical solution to the mapping problem caused by the domain transformation that the algorithm requires. The use of companding to extend the dynamic range of the fixed point numbers may provide answers. It may also be possible to adjust the computations at each of the nodes in such a way that their output is already in the transformed state, and this non-trivial advancement could further simplify the decoder.

Ultimately, the next major step with LDPC codes will be their adoption and use in widespread applications. The power of these codes, and the advances in decoding and encoding performance have put them into the 'practical and useful' realm. The potential for exploiting the parallel structure of these codes should provide a throughput boost that will also enable their use for high bit rate applications in the near future.

Appendix A

The Forward-Backward Algorithm

The Forward-Backward algorithm (FBA, also known as BCJR after its authors), was developed by Bahl, Cocke, Jelineck and Raviv in 1974, and represented a major step forward in computational efficiency for decoding on trellises. The algorithm is used throughout the course of this work and will be presented here.

Consider a parity check equation $f_1(x_1, x_2, x_3)$, a function of three input variables. Figure A.1 shows a trellis diagram associated with $f_1()$. Having a degree of three, we want to determine the three messages that are to output along each edge, based on the three input messages. A simple, but computationally intensive method is to calculate each output one by one. This requires a number of repetitive calculations, and the FBA removes the need to repeat these, resulting in a more efficient procedure.

Each stage in this trellis represents one of the three variables that participate with the particular parity check, and each node represents points where the parity function is satisfied. Edges represent the possible values that each variable an take. Each edge also has a corresponding input message, $u_{f \to x_i}(z)$, represented by $P_{x_i}(z)$ in the figure. Any possible path between the nodes represents a possible combination of (x_1, x_2, x_3) which satisfies parity.

The FBA is aptly named, as it first computes a set of *forward* values, known as the α values, and then a set of *backward* values, known as the β values. There are two forward values, $\alpha_0[n]$ and $\alpha_1[n]$ for each stage of the trellis where n is stage in the trellis. Moving

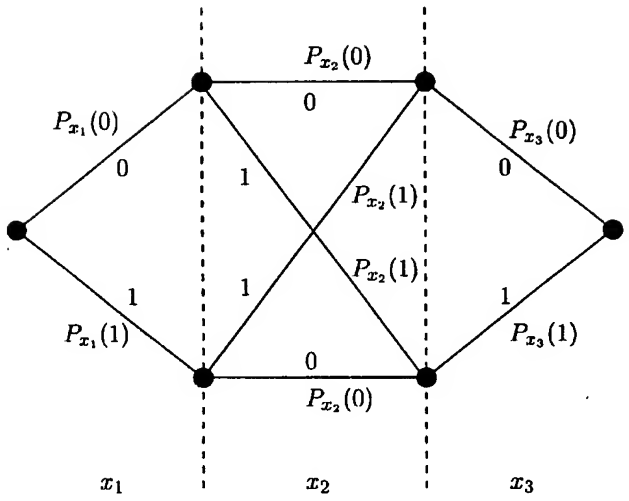

Figure A.1: Trellis diagram of check with sites, x_1, x_2 and x_3

forward through the trellis we calculate the values of α, and $\alpha_0[1]$ and $\alpha_1[1]$ are initialized to 1 and 0 respectively. Figure A.2 shows how α is computed at each stage, and are computed using the equations

$$\alpha_0[n] = \alpha_0[n-1]u_{x_{n-1}\to f_1}(0) + \alpha_1[n-1]u_{x_{n-1}\to f_1}(1),$$

and

$$\alpha_1[n] = \alpha_0[n-1]u_{x_{n-1}\to f_1}(1) + \alpha_1[n-1]u_{x_{n-1}\to f_1}(0).$$

Once all of the α values are determined, there is a backward step, which sweep through the trellis in the opposite direction. Two values, β_0 and β_1 are computed for each stage of the trellis in the backward step. The first values, $\beta_0[3]$ and $\beta_1[3]$ in our example, are set to 1 and 0 respectively. The remaining values are computed using the equations

$$\beta_0[n] = \beta_0[n+1]u_{x_{n+1}\to f_1}(0) + \beta_1[n+1]u_{x_{n+1}\to f_1}(1),$$

and

$$\beta_1[n] = \beta_0[n+1]u_{x_{n+1}\to f_1}(1) + \beta_1[n+1]u_{x_{n+1}\to f_1}(0).$$

After all of the α and β values have been determined, the output messages from each of the check node edges may easily be determined. This is calculated using the equations

$$u_{f_i \to x_n}(0) = \alpha_0[n]\beta_0[n] + \alpha_1[n]\beta_1[n],$$

and

$$u_{f_i \to x_n}(1) = \alpha_0[n]\beta_1[n] + \alpha_1[n]\beta_0[n].$$

At this point, all of the output messages from the node have been determined. This process is repeated at all nodes to update all of the messages in one side of the graph, and this algorithm can be applied for both the check and variable nodes. The algorithm works well regardless of the actual computation required to determine an α or β, whether it is as simple as an addition or a complex product, the overall computational power needed can be greatly reduced, especially as the trellis grows in length.

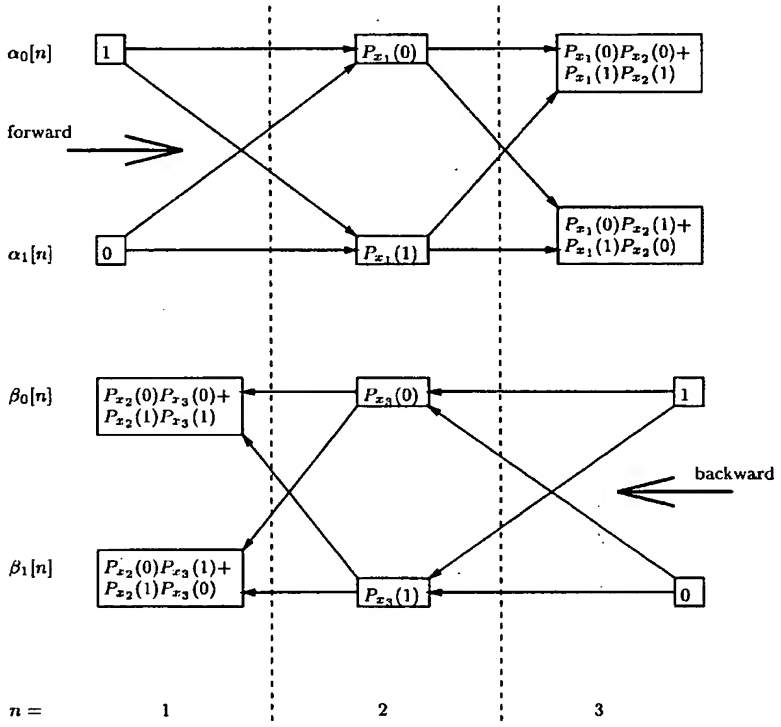

Figure A.2: FBA α, β computation

Bibliography

[1] S. Crozier A. Hunt and D. Falconer. Hyper-codes: High-performance low-complexity error-correcting codes. In *Proceedings of the 19th Biennial Symposium on Communications, Queen's University, Kingston, Ontario*, pages 263–270, 1998.

[2] C. Berrou, A. Glavieux, and P. Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo-codes. In *Proceedings of 1993 IEEE International Conference on Communications*, pages 1064–1070, Geneva, Switzerland, 1993.

[3] T. Cover and J. Thomas. *Elements of Information Theory.* John Wiley & Sons, NY, NY, 1991.

[4] M.C. Davey and D. MacKay. Low density parity check codes over GF(q). *IEEE Communications Letters*, Jun 1998.

[5] M. Frigo and S. Johnson. FFTW: Fastest fourier transform in the west v.2.1.3. http://www.fftw.org.

[6] R.G. Gallager. *Low Density Parity Check Codes.* Number 21 in research monograph series. MIT Press, Cambridge, Mass., 1963.

[7] C. Hartmann and L. Rudolph. An optimum symbol-by-symbol decoding rule for linear codes. *IEEE Transactions on Information Theory*, IT-22(5), Sept. 1976.

[8] R. Johannesson and K. Zigangirov. *Fundamentals of Convolutional Coding.* IEEE Press, New York, New York, 1999.

[9] G. David Forney Jr. Codes on graphs: Generalized state realizations. *Submitted to IEEE Transactions on Information Theory*, 1999.

[10] F. Kschischang, B. Frey, and H. Loeliger. Factor graphs and the sum-product algorithm. *Submitted to IEEE Transactions on Information Theory*, Jul 1998.

[11] F. Jelinek L. Bahl, J. Cocke and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory*, Mar 1974.

[12] M. Luby, M. Mitzenmacher, M. Shokrollahi, and D. Spielman. Analysis of low density codes and improved designs using irregular graphs. In *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing*, pages 249–258, 1998.

[13] D. MacKay. Decoding times of irregular Gallager codes. *In Progress*, Oct 1998.

[14] D. MacKay. Good error-correcting codes based on very sparse matricies. *IEEE Transactions on Information Theory*, 45(2), Mar 1999.

[15] D. MacKay and M. Davey. Evaluation of Gallager codes for short block length and high rate applications. *In Progress*, Jan 1999.

[16] D. MacKay, S. Wilson, and M. Davey. Comparison of constructions of irregular Gallager codes. In *1998 Allerton Conference on Communication, Control, and Computing*, 1998.

[17] P. Hoeher P. Robertson and E. Villebrun. Optimal and sub-optimal maximum a posteriori algorithms suitable for turbo decoding. *IEEE Communications Theory*, 8(2):119–125, March 1997.

[18] T. Richardson, A. Shokrollahi, and R. Urbanke. Design of provably good low-density parity check codes. *Submitted to IEEE Transactions on Information Theory*, Mar 1999.

[19] T. Richardson and R. Urbanke. The capacity of low-density parity check codes under message-passing decoding. *Submitted to IEEE Transactions on Information Theory*, Nov 1998.

[20] K. Gracie S. Crozier, A. Hunt and J. Lodge. Performance and complexity comparison of block-like turbo codes, hyper-codes, and tail biting convolutional codes. In *Proceedings of the 19th Biennial Symposium on Communications, Queen's University, Kingston, Ontario*, pages 84–88, 1998.

[21] P. Sauve S. Crozier, J. Lodge and A. Hunt. SHARC DSP in flushed and tail-biting Viterbi decoders. *The DSP Applications Newsletter*, 41, September 1999.

[22] C.E. Shannon. *A Mathematical Theory of Communications.* Bell Systems Technical Journal, 1948.

[23] Vladislav Sorokine. *Gallager Codes for CDMA Applications.* PhD thesis, University of Toronto, Canada, 1998.

[24] D. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42, Nov 1996.

[25] D. Spielman. The complexity of error correcting codes. *Lecture Notes in Computer Science*, 1279:67–84, 1998.

[26] D. Spielman. Finding good LDPC codes. *36th Annual Allerton Conference on Communication, Control, and Computing*, 1998.

[27] Michael Wall. GAlib: A C++ library of genetic algorithm components. http://lancet.mit.edu/ga/.

[28] S. Wicker. *Error Control Systems for Digital Communication and Storage.* Prentice Hall, Upper Saddle River, New Jersey, 1995.

We present the Sum-X-Sum algorithm, a representation and method for computation of error control decoding metrics, and provide an example for a hardware implementable low density parity check decoder which performs well with low complexity. Performance and complexity are compared with the sum-product algorithm and it is found that while the Sum-X-Sum algorithm performs nearly as well, it is much less computationally complex.

1 Introduction

One of the major drawbacks in the implementation of iterative decoding schemes has been the relative complexity in the decoding procedure. Foe example, low density parity check (LDPC) codes that perform well generally require too many iterations or require block lengths that are too long for practical applications. Conversely, those codes that may be practical don't stand up to comparisons with other decoding methods such as Turbo-codes [1].

It has been a challenge to find a method to reduce the complexity of the iterative decoding process without significantly reducing the performance [2]. We make use of the *duality* found in code graphs to reduce the complexity without an effect on code performance. The use of a dual code for the purpose of decoding is not new [3, 4], but the efficient transformation technique from one state to another has not been previously used.

Iterative decoding operation on graphs generally has two steps; updating the messages sent to checks nodes, and updating messages sent to variable nodes. With the Sum-Product algorithm, a complicated yet powerful decoding technique, it is simple to compute the 'Sum' messages, but quite complicated to compute the 'Product' messages [5].

We solve this difficulty by transforming the messages needed for the 'Product' computation and transforming them into their *dual* representation so that the 'Product' computation actually becomes a 'Sum' computation. The added complexity of the transformation is generally much less than the savings from the dual message calculation. The algorithm may be implemented in conventional manners, both in hardware and software.

2 Algorithm Description

Iterative decoding schemes such as the sum-product algorithm are generally described in a graphical form. Figure 1 gives an example of a graphical description of a parity check code. The graph is bipartite, with variable nodes being represented by the circles at the top, check nodes represented by squares at the bottom, and edges showing the interconnection between the two. Iterative decoders operate by passing messages back and forth along the edges, computing new messages at both the variable and check nodes.

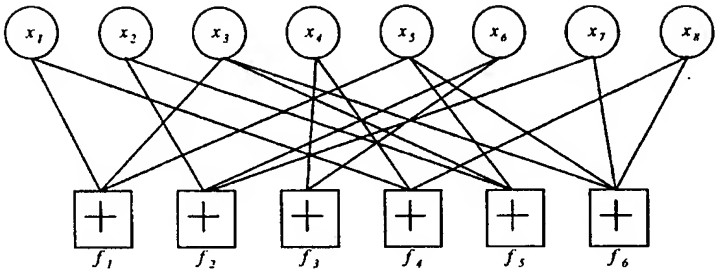

Figure 1: Example of a graphical representation of an LDPC code. The circles represent variables, and the squares are parity check nodes

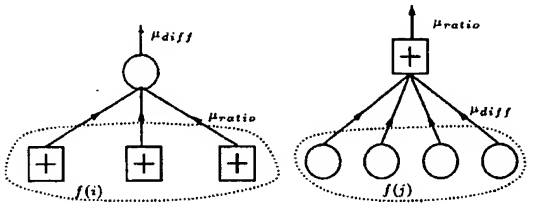

Figure 2: Message passing and updating at variable and check nodes showing which message representations are used for each update equation.

The Sum-Transform-Sum Algorithm (Sum-X-Sum Algorithm), is an instantiation of a soft-input, soft-output iterative decoding algorithm, and is similar to the standard sum-product or min-sum algorithms. The inputs to the algorithm are soft representations of the probabilities for the possible transmitted symbols. For example, over the binary input channel, these possible states are 0 and 1. The probabilities may be represented by a floating point value, a fixed point value or some other proportional means, depending on the actual implementation.

Unlike other iterative decoding schemes, the Sum-X-Sum algorithm utilizes two separate probability representations for messages and requires a transformation algorithm to convert between the two. The method with which the probabilities are transformed may be implemented using logic, functions, memory, or any combination of these.

The two representations for messages used are the log-ratio and the log-magnitude-difference, both of which have been used in other decoding algorithms [2] [6]. The log-ratio representation can be written,

$$\mu_{ratio} \equiv \log \frac{P[0]}{P[1]} \tag{1}$$

The log-magnitude-difference representation is a pair of values (s, r), the first defined over the set $\{-1, 1\}$ representing the sign bit, and the second over the set of positive real numbers such that

$$\mu_{diff} \equiv (sgn\,(P[0] - P[1])\,, |\log|P[0] - P[1]||) \tag{2}$$

With two separate probability representations, the Sum-X-Sum algorithm requires a function $\Gamma(x)(s, r)$ to transform messages between them. Let μ_{ratio} be a message in the log-likelihood ratio domain, and μ_{diff} be a message in the log-likelihood difference pair domain. The function $\Gamma(x)(s, r)$ takes as its argument

a message of the form μ_{ratio} and returns a message of the form μ_{diff}. The returned message has the equivalent probability as the input message, only expressed in a different manner. Similarly, the inverse transform function $\Gamma^{-1}(s,r)(x)$ takes as its input a message of the form μ_{diff} and returns the equivalent probability in a message of the form μ_{ratio}. The function $\Gamma(x)(s,r)$ may be written as

$$\Gamma(x)(s,r) = \left(\text{sgn}\left[2\left(\frac{e^x}{1+e^x}\right)-1\right], \left|\log\left|2\left(\frac{e^x}{e^x+1}\right)-1\right|\right|\right) \tag{3}$$

where x is a message of the form μ_{ratio}. Likewise, the inverse transform function $\Gamma^{-1}(s,r)(x)$ may be written as

$$\Gamma^{-1}(s,r)(x) = \log\left(\frac{1+se^r}{1-se^r}\right) \tag{4}$$

where (s,r) is the sign and magnitude of a message of the form μ_{diff}.

The function in (3) and its inverse, (4), may be implemented very simply using a pair of look-up tables in memory. If, for example, messages are stored as 8-bit (N-bit in general) values, then two look-up tables each of size $2^8 = 256$ bytes (2^N N-bit values in general) are required to completely describe the forward and inverse functions. One need only to calculate the value of the transform for each possible input once, and store the results in memory for future use. Alternatively, the function and its inverse may be used themselves to calculate the necessary transforms on an as-needed basis.

It may also be possible to implement the transformation through the use of a combination of logic and memory. For example, it may be a simple matter to use some logic to determine the most significant bits of the transform output, and determine the least significant bits in a reduced look-up table. This flexibility allows the algorithm to be used in an optimal manner depending on the platform that is it is implemented on. If processing speed is a constraint, it would make sense to take advantage of the look-up table as much as possible as opposed to calculating multiple transforms.

The algorithm passes and updates two sets of messages back and forth iteratively, one set such that the parity constraints are met using μ_{diff} representation as inputs, and the other such that certain variable constraints are met using the μ_{ratio} representation. An examination of Figure 2 shows how messages are input into a node and an output is determined. For each message input into a node, a corresponding message which has been transformed is used to update that message going in the other direction. This will be used as an input for the other side of the graph. For example, the output of a check node will be transformed from a μ_{diff} to a μ_{ratio} message.

Two arrays may be used to store the set of messages being passed in both directions on the graph, one for check node input (and hence variable node output), and the other for variable node input. The message updating rule for a variable will take as its inputs a number of elements from the variable input array, and output in a transformed representation, the same number of elements to possibly different elements of the check node input array. This is done for all variable nodes in the graph so that all elements in the variable input array are used possible a multiple of times.

Likewise, the message updating rule for check nodes takes as inputs a number of elements from the check input array, and outputs in the transformed representation the same number of elements to possibly different elements of the variable input array.

2.1 Algorithm Initialization

Information is sampled from the channel and converted into probability messages of the form μ_{ratio} and are stored in an array as the messages μ_{init}. This set of probabilities are used to determine whether a valid codeword has been received. If a valid codeword is detected, no decoding needs to be done, and the codeword is output from the algorithm. If a valid codeword is not detected, the received messages are transformed into μ_{diff} messages and are used to seed the check node input array.

The algorithm requires a useful description of the code construction, which may be a table listing the connectivity between the check nodes and variable nodes. The actual table may be randomly determined using permutations or some other technique, or the code may be constructed manually. This table is used to determine which messages participate in a given check node or variable update equation, which are the sets $f(i)$ and $f(j)$ in Figure 2.

2.2 Decoding Procedure

Once information has been sampled off the channel and a valid codeword has not been detected, the decoding algorithm commences by updating the variable input array messages. This is accomplished using the following equation:

$$\mu^i_{ratio} = \Gamma^{-1}\left(\prod_{x \in f(i)} s_{\mu^x_{diff}}, \sum_{x \in f(i)} r_{\mu^x_{diff}}\right) \tag{5}$$

where i is the ith element of the variable input array and $f(i)$ is the set of messages connecting to the node associated with i, *excluding* i itself. The variables s and r represent the sign and magnitude for the pair (s, r) which is how μ_{diff} messages are represented. This equation is used a number of times based on the structure of the LDPC code to update all of the elements of the variable input array. In order to increase the efficiency of this equation, the *forward-backward* algorithm may be used [7], a one-pass algorithm may also be used [8], or some other efficient technique may be utilized. These techniques also apply to equation 6 when updating the check input array.

Once completed, the check node input message array is then updated. The rule for updating these messages is governed by the equation

$$\mu^j_{diff} = \Gamma\left(\mu^j_{init} + \sum_{x \in f(j)} \mu^x_{ratio}\right) \tag{6}$$

where j is the jth element of the check node input array and $f(j)$ is the set of messages connecting to the node associated with j, *excluding* j itself. The order in which the variable and check node input arrays are updated may vary depending on the actual code used. The updating may be performed in a serial manner or parallelism may be exploited to increase the overall throughput of the decoder. The actual update schedule may vary, it is important only that all messages are updated in a timely manner.

After all of the messages have been updated once, one iteration of the decoding algorithm has been completed. The decoding algorithm may check to determine if the current probabilities in both the variable input array and check node input array correspond to a valid codeword. If a valid codeword is detected, the algorithm stops and the current probabilities are output. The algorithm may also continue the decoding procedure, by repeating each of the two update steps until some criteria is reached such as a certain fixed number of iterations, or until a valid codeword is detected.

If the decoding procedure fails to produce a valid codeword, a number of actions may be taken. The errors may simply be ignored and the current estimate of the codeword be output. A second outer code may be used to attempt to correctly fix the remaining errors. Some type of retransmission request may be made, or the output may be flagged as to be in error.

3 Performance and Complexity

Figure 3 shows the bit-error performance of the Sum-X-Sum and sum-product algorithms for a regular (3,6) LDPC code. The Sum-X-Sum algorithm performance approaches that of the sum-product algorithm, and

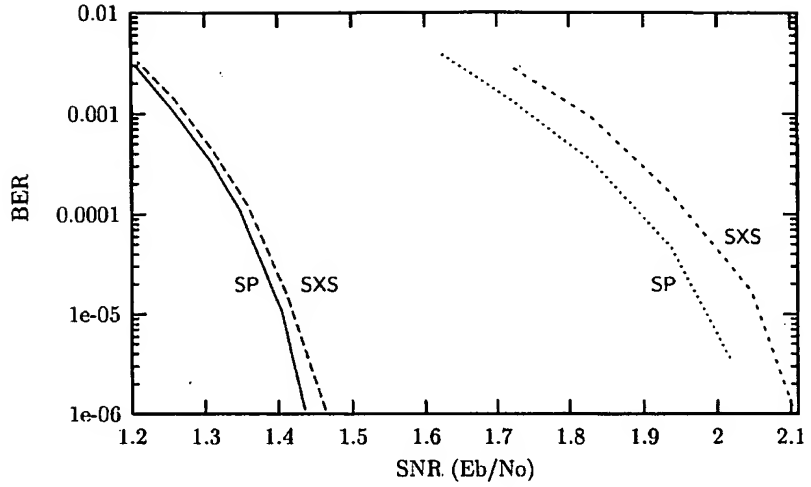

Figure 3: Comparison between Sum-X-Sum and sum-product performance for regular $(3, 6)$ codes. The two left curves are block lengths of 10,000, the two right curves are block lengths of 1000. In both cases the sum-product decoder is more powerful.

for large enough fixed-point precision, the performance actually equals that of the sum-product algorithm. The performance of both these algorithms may be increased significantly by using optimized irregular LDPC codes, but for comparative purposes we shall focus on regular LDPC codes.

Having been designed with reduced complexity in mind, and having seen that the performance of these codes can be quite good, it is important to understand the overall complexity that is involved in the practical implementation of the Sum-X-Sum algorithm. To this end, we calculate the computational power required per bit decoded.

At each computational node, we assume that the Forward-Backward algorithm [7] is used. It is also assumed that the hardware that the algorithm is implemented on is able to handle the bit-depths required with one operation, e.g., 12-bit memory look-ups require one operations, as do 10-bit additions. Taking these assumptions into account, the number of computations required per bit is

$$C_{Sum-X-Sum} = I\left(8 - \frac{5}{\int_0^1 \lambda(x)dx} - \frac{6}{\int_0^1 \rho(x)dx}\right) \tag{7}$$

where I is the number of iterations. It is difficult to do a straight complexity comparison with the sum-product algorithm due to the fixed vs. floating point computation differences. The computations required per bit for the Sum-X-Sum and sum-product algorithms are presented in Figure 4, assuming a one-to-one complexity between fixed and floating point computations. The Sum-X-Sum algorithm requires a significantly lower number of computations than the sum-product, so clearly there is an advantage in its use. Given the pessimistic comparison between fixed-point and floating-point complexity, it is encouraging to see that as a worst case, the Sum-X-Sum algorithm performs quite well for its complexity. A more realistic comparison will only show an overall better outcome in favor of the Sum-X-Sum decoder.

Code	Complexity	SNR for 10^{-3} BER	SNR for 10^{-5} BER
Regular (3,6) SXS	5.33	1.27dB	1.41dB
Regular (3,6) Sum-Product	11.25	1.22dB	1.36dB

Figure 4: Complexity and target SNR for Sum-X-Sum Algorithm codes of length 10,000 vs. similar codes using the sum-product algorithm. Complexity is presented as computations per bit per iteration.

4 Implementation Architecture

The Sum-X-Sum algorithm has been designed with the intention of being easily implemented in a fixed-point hardware environment, and we have seen that with this goal in mind good performance is still possible. The use of a non-optimized, regular code simplifies the structure of the hardware required to implement the algorithm and will be used to describe the hardware. In general however, any iterative code may use this algorithm

Let the code be a regular $\{1000, 3, 6\}$ code, therefore the number of edges on the graph is 3,000. Figure 5 shows a block diagram for a possible implementation, and is the used for this example. Assume that there is a method to convert channel samples into fixed-point probability messages (μ_{ratio} messages), and store the received word in an array called Input. There will be two sets of messages that will be needed, and the memory requirement for these are each of size 3,000 and are the check and variable message memory blocks. There is also the need for two look-up tables to perform the necessary transformations from one message domain to the other, called the check and variable node transform blocks. The memory requirements for these arrays is entirely dependent on the fixed-point precision used, and if we assume 8-bit precision, we require a memory of size $2 \times 2^8 = 512$ bytes.

The decoder must know which edges participate at each of the nodes, and this is achieved through the use of the check and variable node descriptors, both of length 3,000. Each of these arrays contains a list of how each edge is connected to its neighbour, and provides the information required to perform the update rules. Note that there is only a need for one of these arrays, as the information from one can be determined from the other, but the format in which the information is stored in both allows for a simple approach to updating both check and variable nodes.

Once the input array is initialized, the variable message memory block is set to be erasures, and the decoding process may commence. The variable node descriptor is stepped through and in groups of three, the FBA is used to produce three output messages which are transformed and written to the proper locations in the check message memory block. A similar process is used to then write new variable messages. The process repeats until some termination criterion is satisfied, which is not shown in Figure 5. The criterion may simply be some counter tracking the iterations, or may be a test to determine if a valid codeword has been reached. After the decoder has terminated, the decoded word is output. This may be a soft decision or a hard decision, but some decision is made and is output. The word may be used from that point in future processing.

This relatively straightforward implementation requires an overall memory of size $13,512$, or about $13k$. In general, the amount of memory M needed for this implementation for a code of length n and a bit precision of b is

$$M = 4n \int_0^1 \lambda(x)dx + 2^{b+1} \tag{8}$$

It is also useful to look at the throughput capabilities for this architecture. The number of computations per bit per iterations is given in equation 7. Given a rate $R = \frac{1}{2}$ code, and limiting the number of iterations to 25, the number of computations required per information bit is

$$C_I = 25 \times 5.33 \div \frac{1}{2} = 266.5 \tag{9}$$

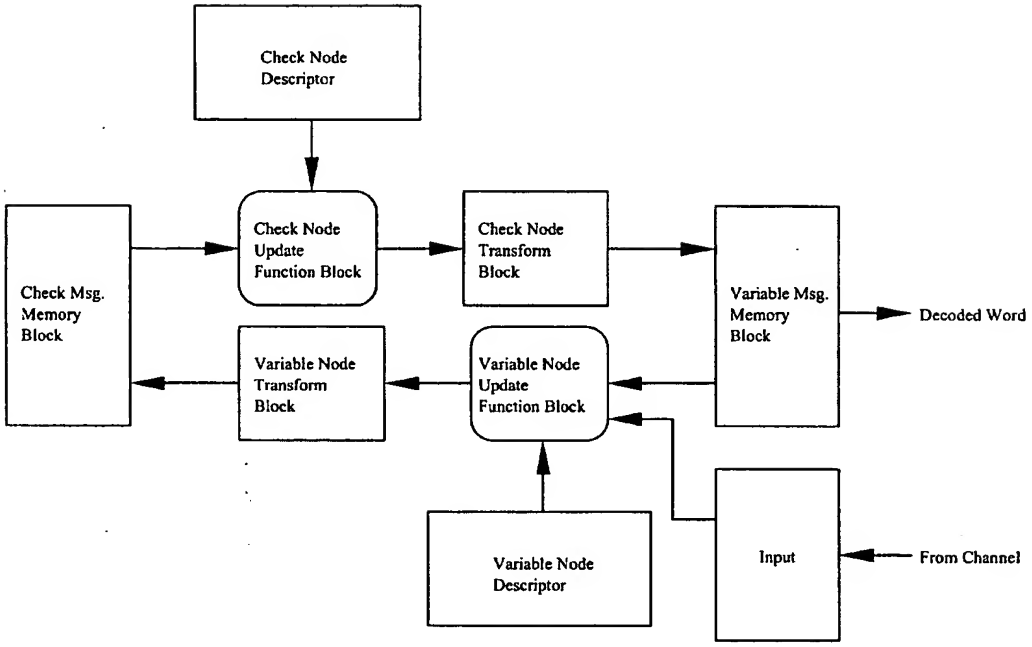

Figure 5: Block diagram of Sum-X-Sum algorithm architecture. Square blocks represent memory locations, rounded boxes are computation nodes.

In a completely serial implementation such as the one presented, if the processor was capable of 40MIPS, then the throughput would be approximately 150kbps. As a very rough comparison, convolutional codes can achieve about the same throughput on a similar system, though with fewer memory requirements [9]. However, these codes do not necessarily perform as well.

One of the benefits of LDPC codes is their inherent parallel structure. This structure can be taken advantage of in a hardware implementation to greatly enhance the throughput of the system. It would not be difficult to introduce parallel update function blocks to speed up the decoder, and although doubling the number of function blocks wouldn't necessarily double the throughput, it would increase it dramatically as most of the computations required take place in these blocks.

5 Conclusions

We presented a novel iterative decoding representation and method which is well suited to practical implementations. The use of the dual domain can greatly simplify the decoding procedure. It is a viable solution for a wide variety of codes, and can perform asymptotically as well as the sum-product algorithm. The overall complexity can be made very low through the use of look-up tables or other simple transformation techniques to perform some of the required calculations.

References

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding: Turbo-codes," in *Proceedings of 1993 IEEE International Conference on Communications*, (Geneva, Switzerland), pp. 1064–1070, 1993.

[2] D. MacKay, "Good error-correcting codes based on very sparse matricies," *IEEE Transactions on Information Theory*, vol. 45, pp. 399–431, Mar 1999.

[3] C. Hartmann and L. Rudolph, "An optimum symbol-by-symbol decoding rule for linear codes," *IEEE Transactions on Information Theory*, vol. IT-22, pp. 514 517, Sept. 1976.

[4] G. D. Forney, "Codes on graphs: Generalized state realizations," *Submitted to IEEE Transactions on Information Theory*, 1999.

[5] F. Kschischang, B. Frey, and H. Loeliger, "Factor graphs and the sum-product algorithm," *Submitted to IEEE Transactions on Information Theory*, Jul 1998.

[6] T. Richardson, A. Shokrollahi, and R. Urbanke, "Design of provably good low-density parity check codes," *Submitted to IEEE Transactions on Information Theory*, Mar 1999.

[7] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Transactions on Information Theory*, pp. 284–287, Mar 1974.

[8] R. Johannesson and K. Zigangirov, *Fundamentals of Convolutional Coding*. New York, New York: IEEE Press, 1999.

[9] S. Crozier, J. Lodge, P. Sauve, and A. Hunt, "SHARC DSP in flushed and tail-biting Viterbi decoders," *The DSP Applications Newsletter*, vol. 41, September 1999.

We Claim:

1. A method for decoding comprising the steps of:
   i) receiving a set of probability messages in a first format;
   ii) updating said set of probability messages based on an operation that compares said set of probability messages with a first-set of code constraints;
   iii) transforming said set updated at step ii) into a second format;
   iv) updating said set transformed at step iii) based on an operation that compares said set of probability messages with a second-set of code constraints complementary to said first-set;
   v) transforming said set updated at step iv) back into said first format;
   vi) repeating steps ii)-v) on said set of probability messages transformed at v) until a desired level of decoding is reached; and,
   vii) outputting said set of probability messages determined at step vi).

2. The method according to claim 1 wherein said first-set is a set of variable node descriptors and said second-set is a set of check-node descriptors.

3. The method according to claim 1 wherein said desired level of decoding is reached when step vi) is performed a predetermined number of times.

4. The method according to claim 1 wherein said desired level of coding is reached when step vi) is performed until said set of probability messages matches a known type of valid code.